(4)

RADC-TR-90-131
Final Technical Report
July 1990

AD-A225 988

THE AMPS

The MITRE Corporation

Bruce Dawson, David S. Day, Alice Mulvehill

Rome Air Development Center
Air Force Systems Command
Griffiss Air Force Base, NY 13441-5700

90 08 27 298

RADC-TR-90-131 has been reviewed and is approved for publication.

APPROVED:

[illegible]

NORTHRUP FOWLER III
Project Engineer

APPROVED:

[illegible]

Raymond P. Urtz, Jr.
Technical Director
Directorate of Command and Control

FOR THE COMMANDER:

[illegible]

IGOR G. PLONISCH
Directorate of Plans and Programs

REPORT DOCUMENTATION PAGE

Form Approved
OPM No. 0704-0188

1. AGENCY USE ONLY *(Leave Blank)*	2. REPORT DATE July 1990	3. REPORT TYPE AND DATES COVERED Final Oct 85 - Sep 89

4. TITLE AND SUBTITLE

THE AMPS

5. FUNDING NUMBERS

C - F19628-89-C-0001
PE - 62702F
PR - 5581
TA - 27
WU - 20

6. AUTHOR(S)

Bruce Dawson, David S. Day, Alice Mulvehill

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

The MITRE Corporation
Burlington Road
Bedford MA 01730

8. PERFORMING ORGANIZATION REPORT NUMBER

N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Rome Air Development Center (COE)
Griffiss AFB NY 13441-5700

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

RADC-TR-90-131

11. SUPPLEMENTARY NOTES

RADC Project Engineer: Northrup Fowler III/COE/(315) 330-7794

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The major product of the AMPS project is the CAMPS System, the purpose of which is to provide a domain-independent base on which to construct application-specific planning systems. The target applications for which CAMPS-based systems are most applicable are characterized by problems that are highly constrained, and that involve the scheduling of tasks and the allocation of resources. CAMPS is the outgrowth of a number of distinct strands in Artificial Intelligence (AI) research, but the core problem solving approach is that of constraint-satisfaction search. The CAMPS collection of tools and predefined domain-independent knowledge structures are used to construct a description of a problem in terms of variables and constraints. The planning process carried out by CAMPS consists of finding a set of values for each of these variables that satisfies all of the constraints imposed by the domain. Two application systems were built using CAMPS: AMPS and EMPRESS-II.

[illegible]

14. SUBJECT TERMS

Artificial Intelligence, Constrained-based Search, Planning, Resource Allocation, Scheduling

15. NUMBER OF PAGES

104

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT SAR

Contents

Acknowledgements

The AMPS project has involved the efforts of many individuals over a period of years. Most prominent among them was Dr. Richard Brown, who developed the original design for CAMPS and was its project leader for most of its duration. Others who have played a role in CAMPS are: Tom Antognini, Matthew Cox, Bruce Dawson, David Day, Eric Gnat, Tim Howells, Leonard Joseph, Dave Krieger, Richard Marcotte, Stuart McAlpin, Alice Mulvehill, Alice Schafer, Ira Smotroff, Ann Tallant and Monte Zweben. Bruce Dawson has been involved in the development of CAMPS the longest, and is the most intimately familiar with its code.

Although this document was written and edited primarily by the three listed authors, we have in some cases taken freely from other AMPS reports and documents. While in the end we are responsible for the content and scope of the report, we owe a significant debt to all of these former contributers to the AMPS project. We also wish to thank Dr. Eric Braude and Dr. Mark Nadel for help in the final editing of this report.

We would like to thank Barry Press of TRW for letting us participate in his knowledge acquisition sessions and introducing us to the planners of the 9th Air Force. These planners contributed valuable domain expertise, particularly Major Edward Ashley and Major David Waterstreet, who spent several days working with the developers here at MITRE.

Finally, we would also like to thank Dr. Northrup Fowler III, for his interest and support during the course of this project. As we begin the development of yet another system based on the CAMPS architecture, we must thank Dr. Fowler for his continued encouragement.

We have learned much in the course of this project about applying Artificial Intelligence to the difficult planning and scheduling problems found in the real world. We hope that the CAMPS architecture will prove useful to other scientists and engineers facing large-scale planning and scheduling problems. We further hope that the introduction of a new, more-portable version of CAMPS will promote its use by other researchers and developers.

1 Introduction

The AMPS project was begun at MITRE in October 1985 with Air Force funding. It began as an outgrowth of MITRE's KRS project, which planned offensive counter-air missions. CAMPS is a knowledge-rich constraint-based planner whose development was intended to contribute to research in the portability of planning systems to new domains, meta-level reasoning, dynamic reactivity to changes, and declarative styles of plan specification. The principal accomplishments of the program consist not only in the progress made on these specific research issues, but also the very integration of and interactions between them. This report puts the CAMPS system in perspective, describes its architecture, recounts the challenges of its implementation, the applications built using the system, reports on lessons learned, and describes the future of CAMPS.

An abbreviated overview of the CAMPS system can be found in the Executive Summary (Sec. 2). Section 3 places the AMPS project and its technical goals in context with other related Artificial Intelligence research. The major architectural elements, their structure and function, are described in Section 4, while Section 5 and Section 6 comment on two of the CAMPS application systems and the lessons learned in carrying out this ambitious project, respectively.

Appended to this final report is a selection of papers written about CAMPS for publication in various journals and proceedings. These articles concentrate on specific research topics touched on by the CAMPS architecture, and are included here for the interested reader.

2 Executive Summary

The major product of the AMPS project is a domain-independent planning system, referred to as CAMPS. Two application systems were built using CAMPS: AMPS and EMPRESS-II. With the exception of Section 5, this paper will be concerned mostly with describing CAMPS, with occasional forays into the AMPS Air Force tactical air mission domain in order to illustrate a point more concretely.

The purpose of the CAMPS system is to provide a domain-independent base on which to construct application-specific planning systems. The target applications for which CAMPS-based systems are most applicable are characterized by problems that are highly constrained, and that involve the scheduling of tasks and the allocation of resources. CAMPS is the outgrowth of a number of distinct strands in Artificial Intelligence (AI) research, but the core problem solving approach is that of *constraint-satisfaction search*. The CAMPS collection of tools and pre-defined, domain-independent knowledge structures are used to construct a description of a problem in terms of variables and constraints. The planning process carried out by CAMPS consists of finding a set of values for each of these variables that satisfies all of the constraints imposed by the domain.[1] This notion, basic to the operation of CAMPS, is described in some detail in Section 3.

The major contribution of CAMPS has been to build from this abstract problem-solving technique a powerful tool for solving real-world problems. To do this a number of AI techniques were brought together, and some new techniques developed and/or refined. These disparate techniques are described in detail in the body of this report, but they are briefly outlined below.

- The adoption of declarative knowledge representation formalisms throughout CAMPS was an important component in assuring the utility of the core system and providing for the easy addition of new, domain-specific knowledge to that core.
- Virtually all of the knowledge in CAMPS is organized around an abstraction hierarchy, known as the *plan-element hierarchy*. This provides all the usual mechanisms supported by such structures, such as inheritance and the definition of new objects by "mixing" together already defined ones.
- Control knowledge in CAMPS can be encoded in what are called *metaplans*. This knowledge is called on to select the most appropriate control strategy given a variety of domain-specific contexts. It is a basic assumption of the CAMPS architecture that the planning process is a knowledge-rich enterprise, and opportunity is taken to enable this knowledge to be captured within the system.

[1] In fact constraints in CAMPS are weighted by their importance and so there are contexts in which the violation of some unimportant constraints is allowed in order not to violate more important constraints. This is described in Section 3 and Section 4.

- Unlike many other constraint-based planning systems, CAMPS develops plans at many different levels of abstraction. This is useful not only to communicate to users at the appropriate level, but as a means for controling the combinatorial search that would otherwise result when trying to solve large, real-world problems.

- CAMPS is a mixed-initiative system. At any time the user can specify as much or as little of a plan as desired. The user can elect to have the system generate all or only parts of the rest of the plan. To best support this give and take between system and user, a rich set of graphical and textual interfaces are provided.

- The system supports re-planning. Re-planning includes the ability to re-plan on the basis of detecting a change in the "external" world. These changes are detected by active monitoring that can be set up via the relational database called *update notification.*

- Not only is the database used to interact in a standardized manner with other programs and intelligence sources, but CAMPS enables domain knowledge to be stored in the standard relational database system, thereby reducing the sometimes significant memory requirements made by conventional AI data structures.

- During automatic planning, selecting values for variables can make use of the local planning context by virtue of constraints posting *restrictions* on that variable. In this way signficant amounts of backtracking can be avoided during constraint-satisfaction search.

- CAMPS supports the use of *hypothetical worlds*. This mechanism enables multiple plans to be generated in order that they may be directly compared, without incurring an inordinate computational cost. In particular, metaplans designed to fix constraint violations can perform possibly quite complex operations on the developing plan which, if ultimately unsuccessful, can be removed quickly and cleanly, without requiring the careful "undoing" of those steps. Often, such "fixit metaplans" will be pursued in parallel (*via* interleaving) until the best procedure is determined.

The above list is meant to give an overview of the abilities and technques available to the developer using CAMPS as the foundation of an application planning system. The rest of this report provides the interested reader with the details of why some of these features are important, and generally how they were developed and implemented in the CAMPS system.

In the immediate future we intend to make the domain-independant core—CAMPS—accessible and usable to planners in the application and research communities.

3 The AMPS Project in Context

CAMPS is the major product of the AMPS project. CAMPS is a multi-faceted Artificial Intelligence (AI) system designed to provide a broad and powerful platform on which large planning and scheduling applications can be built. CAMPS is multi-faceted in that it integrates a number of distinct AI techniques with a rich interface, enabling a "mixed-initiative" interaction between the user and the system. In the chapter following this one we describe in detail the separate mechanisms and techniques that have been developed and brought together for CAMPS. In this chapter our purpose is to place these techniques in a larger context. We will answer questions such as how CAMPS relates to other planning systems in AI, the types of applications for which the techniques used in CAMPS are particularly well suited, etc. Planning in CAMPS is based on the idea of constraint-directed reasoning, and so we begin this chapter with an introductory discussion of this type of problem solving, along with a very simple example. This is followed by a short historical review of planning. We conclude this chapter with remarks on the difficult problems still facing automatic, knowledge-based planning researchers and how some of the ideas deve'›ped in CAMPS' approach these problems.

3.1 Planning and Scheduling with Constraints: An Introduction

The term "planning" in Artificial Intelligence refers to the process by which a plan of action is established in order to achieve some goal. By a *plan of action* we mean simply an ordered set of actions. It is usually quite obvious how to develop a plan to achieve some goal. For example, if one wants to buy something at a particular store, it is a trivial matter to determine the ordered steps by which this goal can be attained. One can easily imagine a plan something like: Leave the house; walk down the street to the store; choose goods and pay for them; return to the house. Indeed, it is often the case that the steps to be taken are so automatic that it is more difficult to state them explicitly than to actually carry them out. For example, if we had really been assigned the task of buying something at a store, would our plan have included the usually automatic action of closing the house door behind us as we leave? Would our plan have specified the need for looking both ways before crossing any intervening streets? While we probably would carry out these steps, it is not clear that we would have explicitly considered them in our "planning process." Indeed, some researchers have argued that we do not even employ such an inter`ıl data structure as a *plan* in the first place.[9] Our inability to unambiguously introspect on our own problem solving skills has proven frustrating to the many efforts that have attempted to automate the planning process. With only sketchy ideas as to how we ourselves approach the solution of such problems, computer scientists have been left with little guidance in their search for methods by which successful plans can be generated automatically. The planning paradigm in AI has been defined partly by the types of problems it attempts to solve, and partly by the methods employed to solve them.

Planning can be distinguished from other tasks in AI by its *synthetic* nature: planning is a constructive process. In contrast, much of AI has been devoted to automating *analytic* tasks, where a set of data is presented and the problem task is to appropriately categorize the data. Medical diagnosis is perhaps the most prototypical example. Planning also starts from a set of data: a description of the world and a goal. While planning may include the categorization of the initial or goal states in the process of generating a plan, the result of the planning process is nothing like a classification. In order to achieve a goal in even very simple domains it is usually necessary that a significant number of actions be performed in a very particular sequence. The "output" of the planning process—a plan—can itself be quite complex.

There have been many proposals for how best to conduct automatic planning. Perhaps the most natural, especially in light of the early work that first established AI, has been to view planning as a search problem in which different orderings of the allowable primitive actions are constructed until the desired goal state is achieved. To conduct such a search requires two important abilities: the planner must be able to detect when a certain ordered set of actions will be successful, and the planner must have some heuristics to guide its search at least approximately in the right direction. Without the former, a planner would have no halting condition, and so would be doomed to search forever for a successful plan; without the latter, a planner would be able to find a successful plan if one exists, but it would likely take *a very long time*. The manner in which these two abilities are established by different planning mechanisms provides a useful way to distinguish alternative theories of planning.

The dominant method used by AI planning systems for testing whether a plan will be successful or not has been to construct an explicit, propositional description of the "world"—that is, the task environment in which the planner's actions are intended to be executed. To determine if a particular plan will achieve the desired goal in this model of the world the planner simulates the execution of the plan and then tests whether the resultant propositional description of the world is consistent with that described by the goal state. In this discussion we will refer to such planners as *model-based* planners. As an example we can take the well-known "blocks world" domain, in which the planner is assigned the task of placing toy blocks in certain relationships with each other. A set of propositions describing one state in such a world might look something like this:

```
(and (on blue-box  green-box)
     (on green-box table)
     (on red-box   table)).
```

In order to determine whether a particular plan achieves a goal, the planner models the effects of each action in the plan by adding to, or deleting from, this set of propositions. It is then a straightforward matter to see whether this description is consistent with a goal state description, say `(on green-box table)`.

The other essential ability for model-based planners is to conduct the search for plans so that the combinatorial costs are controlled. The problem in constructing an adequate plan efficiently for even simple domains is that steps in the plan interact with other steps. In the blocks world example this can be seen if we consider planning for the goal `(and (on blue-box table) (on red-box blue-box))`. If the first step of a plan moves the `red-box` onto the `blue-box` then moving the `blue-box` onto the `table` may be impeded (for example, if the manipulator can carry only one block at a time). Sometimes these interactions are actually desirable, but most often they lead to plans that fail. Generally, this is the major reason that the planner needs to search through many different plans before finding one that actually works.

There are many ways in which model-based planning systems have attempted to control the search process. We will return to a discussion of these methods in the next section of this chapter.

Constraint satisfaction techniques provide another view of how to conduct the planning process (and problem solving, more generally). In constraint-based planning, the world is not modelled directly as a set of propositions. Instead, the important parameters or features that determine a plan are identified and associated with variables, and the relationships that must hold between features in any successful plan are described separately. In this model of problem solving, then, a problem is cast as a set of variables and a set of associated constraints. The constraints require that certain relationships hold between the values assigned to the variables involved in the constraint. A valid solution is any assignment that respects these required relationships. Thus, rather than requiring that a plan be simulated to verify that it is able to achieve the stated goals, the process of generating a plan is itself a means for verifying its correctness.

To see how this works, consider a simple example. For simplicity, we will not provide a particular interpretation for these variables and their values, but it is easy to imagine that the variables correspond to plan steps and that the numerical constraints between them are intended to enforce temporal ordering relationships. We can associate with the three variables A, B and C the following three constraints:

$$A < B \qquad (1)$$
$$B > C \qquad (2)$$
$$A = B - C \qquad (3)$$

Solving the problem (generating a plan) requires finding an assignment of numbers to the three variables such that these constraints are satisfied. For a given domain of values, the problem solving process must search through the possible assignments in some order until all three constraints are true of the values of the variables. Finding these assignments requires, at the outset, that all of the *possible* values for each variable is specified (where this may include the specification of a function that can generate all the possible values). For this example, we make the requirement that the domain of permissible values for each of the three variables is $\{1, 2, 3, 5, 7, 9, 10, 12\}$.[2]

[2]We have chosen these numbers in order to emphasize that a CSP may constrain the domain of its variables in any arbitrary fashion, and of course, they will very often not be numbers but elements from sets of objects appropriate to the domain. If the domain were limited to variables that took on only integer or real numbers, then CSPs could be solved very efficiently by solving the algebraic relationships defined by the sets of constraints. CAMPS does use algebraic simplification whenever possible; but in general, solving arbitrary CSPs must rely on general search techniques.

There are innumerable ways of conducting a search through the set of possible variable assignments to find an assignment that causes no constraints to be violated. Indeed, how this search should be carried out is the central concern in the study of constraint-directed reasoning. Probably the simplest technique used is that of depth-first search with backtracking. In this approach, one variable is picked arbitrarily, say A, and it is assigned a value, also arbitrarily, say 1. At this point all of the possibly applicable constraints are checked to determine if the assignments so far have caused a violation.

If no violation has been detected (as it would not have been in this example so far, since this is the first variable assigned a value), another variable is selected and assigned a value. If the next variable assignment made were $B = 1$, then when constraints are checked, the constraint $A < B$ would be violated. At this point backtracking would commence, first by trying all other value assignments to the last variable assigned. If a value is found that produces no violations, the process continues forward. If, however, no adequate value is found, then the variable is left without an assignment and backtracking moves back to the next previous variable. In Figure 1 the complete search for this example problem is displayed. The solution discovered by this search is $A = 1$, $B = 3$ and $C = 2$.

$$\begin{array}{lll}
\rightarrow A = 1 & \rightarrow B = 1 \hookleftarrow & \\
& \rightarrow B = 2 & \rightarrow C = 1 \hookleftarrow \\
& & \rightarrow C = 2 \hookleftarrow \\
& & \rightarrow C = 3 \hookleftarrow \\
& & \rightarrow C = 5 \hookleftarrow \\
& & \rightarrow C = 7 \hookleftarrow \\
& & \rightarrow C = 9 \hookleftarrow \\
& & \rightarrow C = 10 \hookleftarrow \\
& & \rightarrow C = 12 \hookleftarrow \\
& \rightarrow B = 3 & \rightarrow C = 1 \hookleftarrow \\
& & \rightarrow C = 2
\end{array}$$

Figure 1: The depth-first search tree for the CSP example. The right arrow ($\rightarrow$) denotes the selection of a variable to be assigned, $=$ denotes the assignment of a value to the variable, and the left arrow ($\hookleftarrow$) denotes a constraint violation that leads to backtracking.

One of the first things to note is that there is more than one solution possible. In this example, adding 1 to each of the variable values will preserve the consistency of the solution (but this cannot be repeated, given the explicit limitation placed on the range of permissible values for each variable). Which solution is found by the constraint satisfaction process will depend upon the order in which variables are selected and the order in which their possible values are assigned. Standard constraint satisfaction assumes that every feasible solution is equally desirable. CAMPS allows constraints to be specified that have different *strengths*, or degrees of importance. Including such "soft" constraints increases the expressive power of the system, but also increases the computational cost of solving a problem. We will return to this issue later in this document. (See, for example, Sec. 4.3, Sec. 4.4 and Appendix A.)

Another point worth noting is the relative rigidity of constraint-based planning. In the process of generating a plan with a classical (model-based) planner any number of actions might be generated that will form the resultant plan. In this example we see that constraint-based planning assumes that a certain set of actions has already been established (in this example we have referred to them simply as A, B and C), and it is only their relative ordering that is being "planned." CAMPS has overcome this basic limitation of the CSP formalism by enabling abstract tasks (or goals) to be expanded. This process generates new variables over which the constraint-based planning mechanism works. We discuss this further in Section 3.3 and in Section 4.5.3.

Finally, a point of great importance is that the amount of searching required to find a feasible solution is strongly effected by the order in which variables are selected and the order in which values are assigned. In the example above, if A had been the last variable selected, following B and C, the number of search steps would have been increased from 14 to 85. (And by adding one more constraint, $C > 6$, the number of search steps could have been increased even more, to 410.) Simple depth-first search with backtrack (as illustrated here) has an exponential computational complexity, that is $O(m^n)$, where n is the number of variables and m is the number of possible values for each variable. This presents unreasonable demands on computing time for any but the most limited problems; for larger problems there will have to be some mechanism employed for pruning this search space.

Techniques for pruning the search space consist of imposing an order on the selection of either variables or values. For instance, in the example above with the addition of the constraint $C > 6$, the search could be significantly pruned by simply reversing the order in which values are tried, starting at the largest values and decreasing by one on each constraint violation. The problem is how to enable a problem solving system to automatically derive this desired ordering of values. In the CAMPS system there is a mechanism referred to as *pre-filtering* (see Sec. 4.4.4 and Appendix A) that for most situations manages to order possible values in a desirable way. This is a technique closely related to "Waltz-filtering" (see, for a discussion of this and other techniques: [40], [29] and [15]). Another, more widespread means of pruning the search space is by controlling the order in which variables are selected. This, too, is used to great advantage in CAMPS. In this case CAMPS has largely emphasized a knowedge-rich approach, in which specialized orderings, called *meta-plans*, apply for planning particular types of goals. We will compare and contrast these methods with other approaches in Section 3.5. In Section 4 we describe in detail how these ideas are carried out in the CAMPS architecture.

So far we have limited our discussion of constraint processing to a small example of simple mathematical relationships. As we mentioned before, this same mechanism can be used for planning and scheduling when the problem is described appropriately. For example, variables might take on values that indicate the type of action that will be taken at some point, or at what time an action should be scheduled. This is how CAMPS represents planning and scheduling problems and thereby allows them to be solved using constraint processing techniques. The particulars of how CAMPS supports the formulation of planning problems as constraint satisfaction problems (CSPs) are left until Section 4.

3.2 Controlling Search in Planning

The central problem in planning is controlling the search process. The previous section provided an introductory description of constraint-based planning and how it differs from what we have called model-based planning. Notwithstanding their structural differences, both approaches share the problem of how to focus the attention of the planner on just those parts of plans that are most likely to lead to complete plans that will be successful when executed. In this section we will describe generally the approach to control taken in CAMPS and compare it to related work in planning.

How can a planner choose successive steps in a plan so as to minimize the amount of backtracking? AI planning technology has made some tentative progress towards this goal. Early planners, such as HACKER and GPS developed general search algorithms that attempted to reduce backtracking. HACKER [38] adopted the so-called "linear assumption" as a first step at guiding the search process. The planner attempted to achieve each of the separate aspects of the desired goal state in turn until an interference between steps was detected. At this point HACKER would backtrack to the previous choice point (operator selection) and select an alternate operator, backtracking as far back as was necessary to avoid negative subgoal interactions. While this is a depth-first search strategy, the linear assumption was meant to start the search out with a relatively good initial plan. GPS introduced means-ends analysis, a search technique that combined forward and backward chaining: forward chaining by selecting operators that most reduced the difference between the current state and the goal state, and backward chaining on the pre-conditions of the operators selected.

Both of these approaches improved planning performance over blind depth-first search on small problems, but larger problems remained far too expensive. In general the problem seemed to be a failure to look ahead and see the ramifications of local decisions to those later on in the plan. Specifically, some plan decisions are more important than others, where importance is determined by how difficult it is for the planner to achieve the sub-goal in a variety of ways. Such constraining plan decisions should be made earlier than others so that the less important steps need not be re-done. ABSTRIPS [33] was a planner that established an explicit ranking of the importance of different types of sub-goals in a plan. By planning for highly ranked sub-goals first, ABSTRIPS managed to avoid significant amounts of backtracking.

NOAH [34] was the apotheosis of this line of research, and was the basis of many subsequent planning systems over the following decade. In NOAH, not only is there a hierarchy with respect to goals, but operators themselves are hierarchically organized, so that planning at successive levels of detail introduces as little ordering among plan steps as possible. Only very late in the planning process are plan steps linearized—the partial ordering is replaced by a completely ordered sequence of plan steps. Since it is generally the relative ordering of plan steps that introduce negative interactions among them, delaying a commitment allows the planner to order the plan steps "correctly" from the beginning, with less backtracking. NOAH and its many descendents are known as *non-linear, hierarchical planners*. (Examples include NONLIN [39],[3] SIPE [43], TWEAK [10], among others.)

[3]Actually, NONLIN is not a descendent of NOAH, but was developed by Austin Tate independently and

In order to know when to introduce linear ordering in the developing plan, NOAH-style planners must provide "critics"—specialized modules that determine when certain types of plan steps should be ordered. In general, it is still difficult in these planners to maximally influence the selection of operators and their ordering as a function of their possible interaction with subsequent plan step choices.

MOLGEN [35] was the first constraint-based planning system.[4] Its strength derived from its ability not only to minimize the degree to which the planner committed to certain plan steps (so-called *least-commitment planning*), but the way in which each minimal commitment to a plan step provided additional constraints for other plan step choices. The restrictions these constraints entail are propagated to all the other variables. Propagating constraints can be viewed as a method of pre-ordering the selection of variable values. In the planning process, as a variable's value is set, then any other variables that are involved in constraints with the original variable have available to them important information. For example, referring back to our mathematical example earlier in this section, if there is a constraint $A < B$, and A has already been assigned the value 4, then there is no need to consider generating the values 1, 2, 3 and 4 as possible assignments for B, since they would clearly violate this constraint. The effect of "propagating a constraint" in this case has the effect that the process of generating values for B effectively excludes these values. In CAMPS this process is called "posting restrictions" and is discussed in more detail in Sections 4.4 and 6.4.

CAMPS is based on MOLGEN's notion of constraint satisfaction problem solving employing constraint propagation. Propagating constraints is one of the most powerful means for limiting backtracking in a domain-independent manner. This technique is particularly useful in planning, since, as we have noted, backtracking is brought about in planning by unanticipated interactions between plan steps. By propagating constraints, a planner is able to dramatically decrease backtracking, since these interactions are in some sense anticipated, and therefore avoided. CAMPS brings to this general model of problem solving a number of mechanisms designed to significantly improve the construction and performance of constraint-based planning systems.

3.3 Enriching Constraint-based Problem Solving

CAMPS is a general constraint-based planning system. It is domain-independent and designed to support the development of planning systems for a very broad range of tasks. Particular emphasis has been placed on scheduling and resource-allocation types of problems. Because of this design philosophy, generating a particular planning system with the CAMPS core requires little to no programming.[5]

roughly contemporaneously with Sacerdoti's NOAH.

[4]It was not the first constraint-based *problem solving* system, however. That distinction probably goes to Richard Fike's REF-ARF system [17] for solving certain mathematical problems with constraint problem solving.

[5]The extent of required programming depends strongly on specifics of the domain and the intended run-time efficiency of the planner, among other things. Details are given in Section 6.

There are a number of ways in which developing a domain-specific planner from CAMPS has been made easy. First of all, CAMPS views the variables of standard constraint satisfaction problems as slots in a hierarchical knowledge base. Instead of attempting to describe a complex planning domain in terms of a "flat" list of variables and associated values, CAMPS allows the developer to define a rich hierarchical tree of concepts and concept instances. This basic and powerful tool of Artificial Intelligence enables a domain to be described in a structurally appropriate manner. This hierarchical knowledge base is called the *plan element hierarchy* in CAMPS. (See Sec. 4.2.1.)

The plan element hierarchy supports inheritance. This adds an important inference mechanism to the standard constraint-processing repertoire, reducing the need for system builders to use awkward constraints to express basic structural relationships between elements of the hierarchy. It is in this hierarchy that the system developer can establish hierarchical relationships between tasks and subtasks, for example. As we have discussed before, hierarchical planning was used in model-based planning to delay commitment to the ordering of plan steps. By supporting task/sub-task relationships in the CAMPS plan element hierarchy, the same advantages can be added to a constraint-based planner as well.[6]

Supporting the ability to describe task/sub-task relationships is an important facet of the CAMPS architecture, for it serves to broaden the applicability of CAMPS-derived systems beyond standard scheduling problems. While there is some ambiguity associated with the terms *planning* and *scheduling*, it is generally agreed that scheduling refers to the process of assigning times to specified processes (actions, events). The original selection and relative ordering of those processes is presumed to have been taken care of by some preceding planning process. CAMPS enables the complete integration of both planning and scheduling in this sense, for the selection and expansion of tasks and their constituent actions (or sub-tasks) is explicitly controlled in CAMPS's constraint-processing approach to problem solving.

This can be contrasted with many other constraint-based problem solving systems. For example, the ISIS system [19, 18] is a constaint-based scheduling system that addresses similar issues as those of CAMPS, especially with respect to its use of AI knowledge representation techniques and weighted constraints. But ISIS explicitly sidesteps the planning part of the problem (alloting this task to a pre-processing search phase) and concentrates instead on other issues in constraint-processing, such as constraint relaxation.[7]

It was a design goal in CAMPS that a system developer need write almost no computer code at all when constructing an application system with CAMPS. Besides variables (defined in CAMPS as slots of elements in the plan element hierarchy), the definition of a planning problem as a constraint satisfaction problem requires that constraints be defined, and that variables be associated with *generators* that generate possible values in light of any propagated constraints. CAMPS provides a host of pre-defined constraints and associated generators. Only rarely will the system developer have to write non-declarative computer code to

[6]This is related, but not identical, to the notion of hierarchical *variables*. MOLGEN used certain variables to describe parts of a problem in more general ways, further extending the delayed commitment to a particular plan ordering. These types of variables can be defined in CAMPS as well. See Sec. 6.4.

[7]Examples of other scheduling-specific systems include NUDGE [23] and [21].

implement specialized constraints; generally all constraints can be written as straightforward rule-like structures, using the primitives provided within the CAMPS library. The details of how constraints are defined are described in Section 4.3.

3.4 Planning under Uncertainty

The general constraint satisfaction formalism is usually restricted to constraints that return only success or failure: a constraint is either violated or it is not. Planning in real-world domains is not usually so cooperative, however. The effects on the world of executing some action are not known with certainty. More importantly, in many domains it is virtually impossible to generate plans with no constraint violations at all (the domain is said to be "over constrained"), and the issue in planning becomes which constraint violations are "allowable," and which aren't. In these situations the system designer will want to be able to express *to what degree* one constraint is important (or desirable) relative to other constraints. A natural approach to this problem is to allow for constraints to be associated with real-valued weights.

To make CAMPS a tool for generating plans in such domains required generalizing the otherwise restrictive form of constraint satisfaction. Constraints are divided into classes: *feasibility*, *efficiency*, etc. Any individual constraint is identified as enforcing a relationship of one of these types. The certainty with which the violation of a constraint leads to, say, the infeasibility of the plan, can be described by a number between zero and one. While we will not describe the details of this scheme here (see Sec. 4.4 and Appendix A), it should be understood that this provides the CAMPS-derived planner with a much greater flexibility than one based on the original formulation of constraint satisfaction problems. In particular, it allows a number of weak constraints to override some single stronger constraint. Any constraints that are absolute can be so indicated (with a weight of 1 or 0), so the developer need only make use of weighted constraints in situations where it is appropriate. Of particular interest is the fact that the generators associated with variables (which provide possible values for a variable when one is desired by the planner) make use of these weights to select the "best" value. (See Sec. A.2 for details.)

Uncertaintly places another restriction on a planner: uncertainty establishes the possibility that a planner's plans might fail, and so if the planner is to continue to pursue its goals, it must be able to generate a new plan. This is called *re-planning*, and is a unique feature of the CAMPS system as compared to any other constraint-directed planning system. We will discuss re-planning more in the next section.

3.5 Meta-Planning

Ea[illegible] in Section 3.2 we described how propagating constraints in a constraint-based system ca[illegible] provide a domain-independent planner with a significant amount of information about the interaction between plan steps so that backtracking is minimized. This does not completely obviate backtracking, however. Backtracking may still be necessary, for when variables are assigned values early on, they are still under-constrained due to the fact that other variables have not yet been assigned values.

The design of CAMPS is predicated on the belief that planning is a knowledge-rich enterprise. The classical formulation of planning has been shown to be NP-complete; that is, it is probably intractable. [9] It follows from this that planning performance can be dramatically improved only by providing mechanisms by which control knowledge can be easily encoded and applied in both general and very specific planning situations. For example, when constraints are violated, very often it is possible to define application-specific methods for debugging the variable assignments made so far. Such methods can be defined along a continuum of generality, including some appropriate to what have been called *generic problem types*. (See, for example, [8] or [24].) Only when there is no specialized knowledge available need one take recourse to generalized ordering and/or backtracking methods. CAMPS attempts to facilitate the application of knowledge to control planning in a number of ways.

CAMPS enables the user to describe specialized control strategies. These control strategies specify the order in which variables are to be assigned values. By supporting multiple control strategies, specialized for particular contexts, CAMPS need not be restricted to a very general, "weak method." Instead, any control knowledge available to the knowledge engineer can be easily incorporated into CAMPS' (or a CAMPS-derived system's) control regimen. This control information is encoded in *meta-plans*.[8]

As an example, for certain tasks in the Air Force tactical mission planning domain, it is known that the fewest constraints are violated if the "time over target" is established first when planning a particular type of offensive counter air mission. This control knowledge is retrieved by CAMPS when it begins to plan for such a task. At that time any and all applicable meta-plans advertise their availability. CAMPS chooses the meta-plan with the highest ranking (which is usually the meta-plan that applies most specifically to the current task at hand).

The notion of specialized control knowledge—or *meta*-knowledge—was first introduced by Davis in TEIRESIUS [11]. Later, Wilensky [42] applied the idea specifically to planning. Meta-planning is a necessary component for any large-scale planning system, for only weak-methods are general enough to solve problems independent of the application-specific knowledge, but weak-methods are simply too slow to solve large problems in a reasonable amount of time. Meta-control knowledge enables the architecture to support more domain-specific knowledge without sacrificing the generality of the underlying architecture or its default problem-solving methodologies.

The movement towards greater autonomy in AI planning systems is imperative, and this is especially so for systems that are to be operated in a dynamic environment. In designing CAMPS this issue was of particular importance, and meta-planning plays a critical role in this regard. The classical view of planning ignored issues in re-planning until recently; recent

[8]These are called *meta-plans* because they provide knowledge about *how to control* the planning process; they do not encode knowledge about what kinds of particular plans should be considered.

work that addresses these concerns has tended to emphasize the minimal restructuring of the current plan, and less the dynamic responsiveness of the planner with its environment. (See [44], [1].) Far more important in our view is the means for responding to changes in the environment quickly (relative to the domain) and intelligently. Here "intelligently" refers both to the desirability for specialized re-planning knowledge, and also to the applicability of real-time constraints.

CAMPS supports a large variety of what are called *fixit* meta-plans. These meta-plans can be triggered by any number of conditions that occur during planning, including changes in the world. In particular, they can provide a problem-specific approach that can be much more efficient due to this specificity. Consistent with the declarative approach throughout CAMPS, these meta-plans can be user-defined with little programming. Other work has shown the power of specialized debugging knowledge, including such early work as Sussman's HACKER and Sacerdoti's NOAH, and continuing through to McDermott's ground-breaking work on planning and action [30] and Wilkin's SIPE [44]. What has only recently become a matter of concern, however, is how to provide planning systems with sufficient knowledge about their real-time constraints that they can select planning techniques with the appropriate processing time/plan quality tradeoff. This approach to control has been referred to as *approximate processing* [28]. A number of recent planning systems have demonstrated the necessity and power of this kind of meta-planning. (See, among others, [14], [22] and [27].)

It would be impossible to provide a "complete set" of approximate processing meta-plans for a domain independent planner—that contradicts the basic assumption that knowledge about the dynamics of a particular environment can and should be exploited by the planner—but CAMPS provides the declarative support that allows system developers to easily encode this knowledge in a declarative fashion. In a later section of this document we will describe two application systems built with CAMPS in which the use of dynamically-triggered *fixit* meta-plans played a prominent role. (See Chapter 5.)

3.6 Interacting with the World

CAMPS was built to support the development of "real world," large scale planning and scheduling problems. Large planning systems will require large knowledge bases. The size of these knowledge bases will generally not be sustainable using the standard AI practice of loading it all into the memory of one appropriately large computer. Even with virtual memory, the performance degradation becomes untenable. Only recently has AI begun to look at the integration of large, efficient external databases. CAMPS was developed from the beginning with these issues in mind, and the underlying mechanisms of the planner make a distinction between knowledge that is "in working memory" (where this is the working memory of the machine, not the AI notion of working memory analogous to psychological models of "short term memory") and knowledge that is in the database (on disk). Constraint operations are defined so that as little as possible of the knowledge base need be pulled into working memory. The details of this mechanism are described in Section 4.8.

3.7 Summary

As we noted at the beginning of this section, the CAMPS planning system is multi-faceted. Some of what makes CAMPS a unique system is due to its having succeeded in bringing together a number of different strands of AI research and weaved them into a coherent, general and potentially quite powerful core planning system. Along the way a number of technical hurdles had to be addressed. Our successes in solving those problems, some minor and some more significant, are steps of a process that is of growing importance to AI: bringing the fruits of disparate and very original technological advances together into working systems for large, real-world applications.

In the following chapters we will describe in much greater detail how the CAMPS architecture is structured and the way in which it is used to describe and solve different problems. We will also relate to the reader our experiences in using CAMPS. Over the development history of CAMPS much more was learned about planning and scheduling than what is exhibited in the current version of the CAMPS program.

4 The CAMPS Architecture

In this section we begin by providing an overview of the CAMPS architecture, along with some discussion on how the elements of this architecture interact to provide a comprehensive planning, scheduling, and resource management environment. More detail is then presented on the major elements of the CAMPS system. A brief outline of this section should prove useful to the reader. Section 4.1 gives an overview of the system and how it can be used. In Section 4.2 the reader is introduced to the major data structures around which the CAMPS architecture is constructed. CAMPS is, at its core, a constraint-based planning system, and in Section 4.3 the various types and aspects of constraints are described. Constraints establish that certain relations between variables be maintained. In CAMPS these variables are complex structures enabling them to support many of the AI and other techniques brought together in the system. The structure and function of variables in CAMPS are described in Section 4.4. (One part of variables—*filter trees*—are treated in much more detail in Appendix A.) The way in which *metaplans* are used to control the planning process is described in Section 4.5.

These sections will have given a largely complete introduction to the main parts of the CAMPS planning system. In Section 4.6 we present some of the issues that arise in scheduling and how CAMPS approaches these problems, including the treatment of resources and their allocation. The use of hypothetical worlds in CAMPS is described in Section 4.7. CAMPS is somewhat unique in its use of a relational database system to store knowledge and interact with the "external world." This is discussed in Section 4.8. Finally, the CAMPS interface is described in Section 4.9.

4.1 Introduction

CAMPS is an AI-based problem-solver that incorporates constraint-based planning and meta-planning techniques. In CAMPS a planning problem solution can be viewed as a set of tasks, the resources those tasks will use, and the schedule of tasks and resource utilizations such that all constraints are satisfied. The CAMPS architecture is a formalism and a set of inference mechanisms in which all relevant facts about this type of planning application can be declaratively represented. These "facts" include definitions of application-specific terms and relationships among these terms, correctness and desirability criteria, and problem-solving strategies (metaplans). CAMPS has been implemented in two domains, the Air Force application of CAMPS called AMPS and the NASA/KSC application of CAMPS called EMPRESS-II. These applications will be discussed in Section 5. In this chapter we concentrate on the domain-independent core of those two application systems, which is CAMPS. From time to time we will draw on the AMPS domain to illustrate architectural elements with concrete examples.

The CAMPS architecture consists of the following architectural elements (see [5] and [6] for details):

- A *knowledge representation formalism* in which all application-specific knowledge can be expressed declaratively as constraints, rules, plan element objects and database entries.

- A *meta-knowledge formalism* in which both general and application-specific problem-solving strategies can be expressed in a predominantly declarative fashion, called *meta-plans.*
- A *relational database* to contain ever changing situational data. This database is shared among planning "workstations," intelligence analysis tools, and activity monitoring tools (update notification).
- A *working memory* that contains the tasks, resources, and other information being used and manipulated by the system. Working memory is dynamically filled from the relational database, and consistent partial results can be stored back into the relational database.
- A *knowledge base* containing all the knowledge and the declarative portion of the meta-knowledge specific to an application, e.g., the plan element hierarchy.
- A *user's toolkit* (e.g., tables, graphics) with which to explore, display, and modify the relational database, working memory, and the knowledge base.

Within this architecture, several facilities are uniformly supported, such as:

- *Belief/disbelief evidence combination.* This is used for four different purposes: (1) to "soften" predicates and allow evidence to be flexibly combined; (2) to dampen out inconsistencies among the rules in the system; (3) to quantify "how well" a suggestion satisfies a number of constraints; and (4) to provide an a priori estimate of how successful some suggested change will be in solving a planning problem.
- *A uniform record of justification is maintained.* This is used for three different purposes: (1) to provide explanations of how and why the system made choices; (2) to assist in deciding which selections and choices to replan; and (3) to avoid needless duplication of effort.
- *Hypothetical worlds.* This capability provides the user and the metaplanning component of CAMPS with an environment in which to hypothesize and test assumptions about existing or anticipated conditions of the environment, e.g., weather, or about the relevance of slot values, e.g., the best aircraft to use to fly a particular type of tactical fighter mission.

4.1.1 Using CAMPS as a Planning Tool

The figure "CAMPS System Architecture" (see Figure 2) is a high-level picture of the ways in which CAMPS uses various declarative sources of knowledge: specifically constraints, rules, and metaplans. Plan elements have constraints attached to them; these constraints are checked as the plan element slots are being filled. Constraints are stated in terms of predicates. To see if some predicate is true, the rule interpreter may be invoked. This interpreter uses a mixture of rules from a library and predicates directly defined (some in terms of the relational database) to determine if predicates are true or not. The same rules are also used to help figure out what could be changed in the plan element knowledge base to make some predicate TRUE or FALSE.

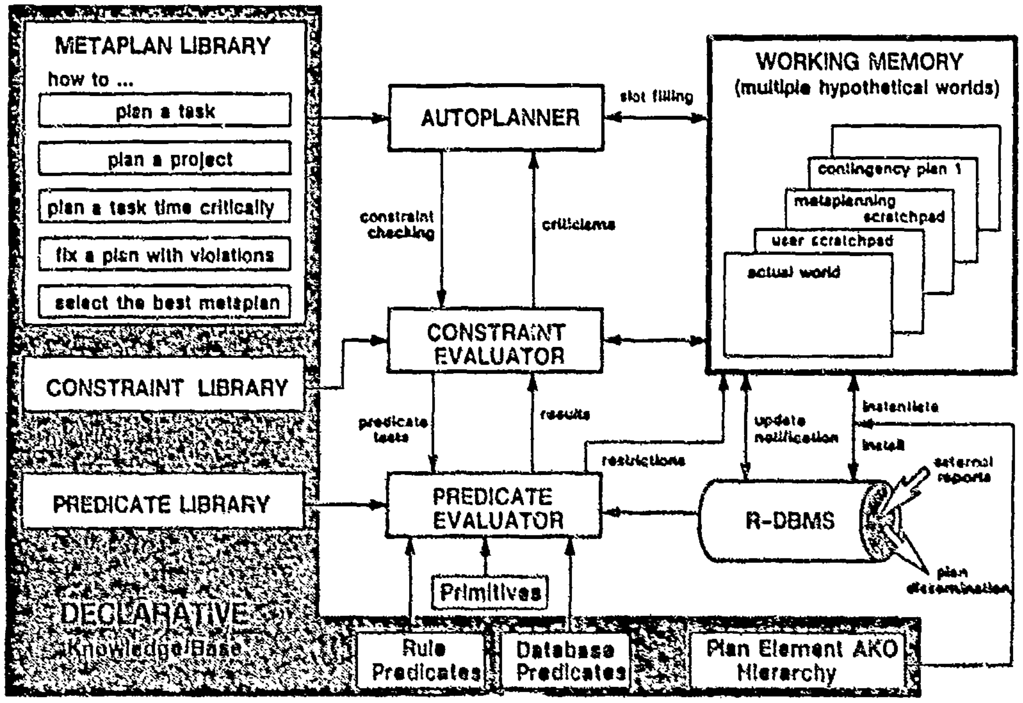

Figure 2: The CAMPS Architecture

CAMPS solves planning problems for the user by trying to achieve certain metaplanning goals. Modern planning systems usually distinguish the "what is" knowledge that captures the salient features and constraints of a planning application from the strategic reasoning that effectively applies such knowledge to accomplish some goal.[12] [41] [36] The latter, typically termed meta-rules or meta-knowledge, provide an explicit and extensible representation of the control strategies required for intelligent planning. The CAMPS architecture includes a facility that lets the user formally express some of these goals, thereby allowing CAMPS to use its problem-solving skills in the user's behalf. Although CAMPS can plan with little or no intervention from the user with use of it's metaplans, all decisions are ultimately the responsibility of the user, and thus under user control [26].

The CAMPS metaplanning component [3] provides a mechanism for posting goals to the system and utilizing a mix of declarative metaplans and procedural standard control flows to accomplish goals. When new subgoals are posted, new metaplans are instantiated to accomplish these subgoals. The resulting hierarchy of active problem-solving agents provides global control over local planning actions (e.g., filling in a slot or checking a constraint).

A central feature of this approach to metaplanning is the separation of the meta-knowledge into two parts, and the use of an active agent constructed from those two parts to actually implement a problem-solving strategy [5]:

Metaplans are declarative meta-rules that can be easily manipulated, modified, and explained by or to the user. Metaplans specify what should be done to accomplish a specific goal, but refer the issue of how that metaplan should be implemented to the agenda.

Agenda flavors are standard control and data flows. They constitute an easily extended vocabulary for problem-solving. When one casually says "I use a generate-and-test algorithm to select a consistent pair of X and Y," one has said both *what* ("select a consistent pair"), and *how* ("use a generate-and-test").

Agenda instances are active objects with various pieces of code (flavor method handlers) that serve to implement a specific control and data-flow. They act as agents that know how to carry out the "what" specified by a metaplan.

The problem-solving agents provide CAMPS with a high-level, top-down view of planning and resource allocation. However, problems with planning usually arise because some detail is out of place. This defect in a plan is signaled to the metaplanning component via a constraint violation. CAMPS provides three modes of constraint evaluation, one of which, *make-mode*, is intended to provide a low-level, bottom-up view of the planning problem by producing a structure that suggests some action the problem-solver might take to resolve the problem and eliminate the constraint violation [26].

The ability to respond to unforeseen conditions in the environment (replan) is another major design goal of CAMPS. In order to achieve replanning capability, problem-solving strategies and predicates must communicate in an orderly manner. Metaplans typically suggest ways of undoing some planning decision in order to retain consistency with some unforeseeable change in the planning environment.

Throughout the CAMPS session, the user and the knowledge-based system try to work together in a mixed-initiative mode where the user makes changes, CAMPS reports on new conflicts, the user repairs some conflicts and asks CAMPS to take care of the rest. Any decision CAMPS makes can be later reviewed, questioned and changed by the user. Any decision the user can make can also be delegated to CAMPS. CAMPS finds all inconsistencies in plans being developed, but does not insist that the user remove inconsistencies (except for errors of typing, e.g., using a name where a number is expected or using a task where a resource is required). Finally, CAMPS knows who made what decision, and will not change a user decision without at least telling the user, and generally asks the user's permission first.

CAMPS also provides a hypothetical planning environment by which users and/or metaplans can evaluate a series of candiate plans all somewhat appropriate for accomplishing a goal. CAMPS uses a scheme similar to [16] to tag the hypothetical world in which a slot is filled with a particular value. Alternative hypothetical worlds can be maintained and a plan-element's slots can contain different values in different alternative hypothetical worlds.

In order to support planning in an environment where different aspects of the planning problem are solved by different people (different agents are responsible for some defined set of slots), CAMPS provides a foundation by which [illegible] support a multiple workstation environment. Although support to multiple workstations has not been implemented, the necessity of supporting this planning environment can be easily understood by viewing how planning is performed in one of the application domains to which CAMPS has been applied. In AMPS, plans are generated to obtain targets through the concerted errorts of a number of agents (human planners). For example, intelligence officers provide intelligence information about the existance and persistance of targets; officers at the wing level track the allocation and usage of aircraft and pilots; and at the planning staff level plans are specified for creating missions to hit delegated targets in a given time period with respect to the Commander's guidance of the day. One would envision a workstation for each of these planners.

Within a multiple workstation environment, different planners will want access to much of the same information as other planners. CAMPS uses a database management system (DBMS) to provide a single, uniform interface for shared access to the data, even within a distributed environment. Some DBMS allow concurrent modification of shared data by multiple users. These multi-user DBMS make this interface available concurrently to multiple users. The multi-user DBMS [illegible]ures that the data are not corrupted and that incorrect results are not produced due to inappropriate interleaving of data processing operations from different users. It al[illegible] prevents users from being permanently deadlocked (two or more users holding data resources while waiting for one another to release held resources) or from being continually aborted and restarted.

4.2 The Structure of Plans in CAMPS

4.2.1 The Plan Element Hierarchy

A long-standing property of AI reasoning systems is that of generalization and specialization through "a kind of" (AKO) hierarchies. Indeed, such hierarchies are almost a trademark of systems based on AI. We use the term "plan element" to refer to the most general notion of "the things CAMPS deals with." A plan element is thus an "object" in object-oriented working memory. Each plan element is represented in working memory in a frame-like manner. The attributes, parameters, and cross-references among plan elements are stored in the plan element's slots. Plan elements in working memory are created from information in the database by a process called instantiation, and [illegible] stored into the database by the installation process.

In CAMPS, each class of plan elements is defined by a set of capabilities. A capability itself is further defined as a set of capabilities (called the components of the capability), so that the plan element class is the set of capabilities, their components, the components' components, etc. The capabilities thus form a so-called tangled AKO hierarchy ([5] [6]) (see Figure 3).

Each "capability" has associated with it a set of "slot descriptions." The set of slots associated with an individual instance of a capability is the union of the slots associated with

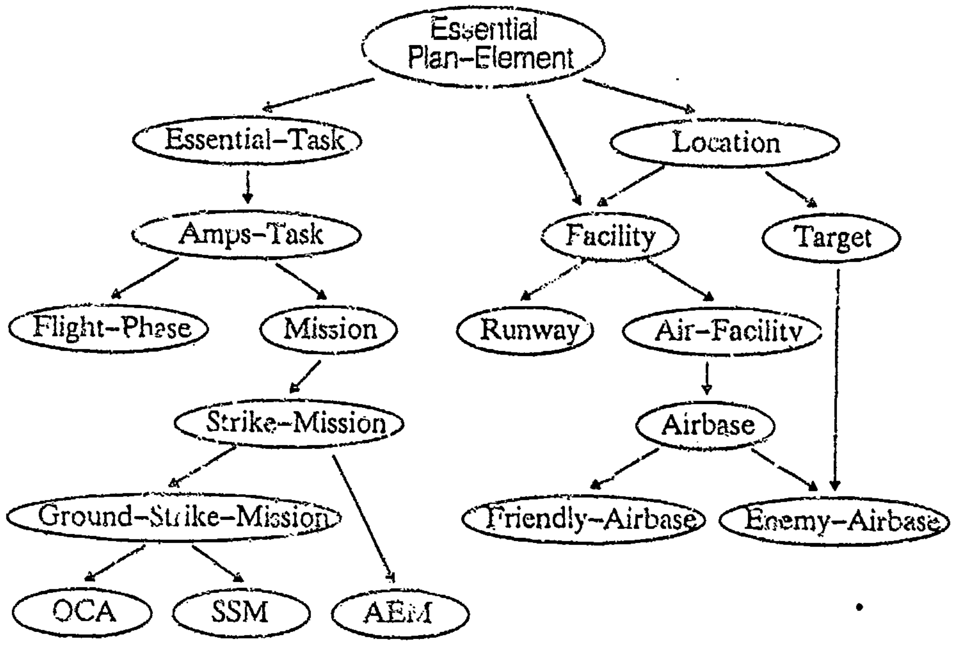

Figure 3: A partial view of the CAMPS AKO hierarchy.

the capabilities of the class. Each instance of a plan element capability, when completely planned, will have all of its own individual slots filled.

In our implementation, "capabilities" and "plan element classes" are implemented by ZetaLisp flavors augmented by additional descriptive information, and "plan element instances" are implemented by flavor instances. The mixing of "capabilities" is thereby implemented by the flavor mixing facilities in ZetaLisp. Thus the CAMPS AKO hierarchy is derived from the underlying flavor hierarchy, which is itself supported by the ZetaLisp dynamic type checking facility supported by special hardware in some LISP Machines.[9]

4.2.2 Viewing Planning as Constrained Slot-Filling

Since all parameterization and inter-relationships among the plan-elements (i.e., a plan) are represented by the contents of each instance's slots, planning can be viewed as filling slots subject to constraints. A "slot" is further annotated with the location where the actual information for the slot is stored in working memory (e.g., locally with the instance, indirectly in another slot, or with the class).

[9]This and other implementation details may change as work moves forward on a more portable version of CAMPS to be implemented in Common-Lisp.

There are three types of slots in CAMPS; remote, local (single-valued and set-valued), and indirect. The importance of indirect slots must be recognized. A plan element instance is a local planning context comprising that instance's slots, each of which is a decision variable. That slots can be "indirect" means that these local planning contexts overlap, and that the overlap is determined dynamically by what plan element instances fill slots.

The CAMPS architecture allows the user to freely change the value of any slot at any time (provided they have "modify access rights"). Of course, doing so may wreck a plan; users are allowed to express arbitrarily bad and inconsistent plans. Furthermore, constraints (described below) can always be checked in a mode (normal-mode) in which no side-effects on slots will occur. Thus the values of slots are under the user's control.

At the implementation level, information other than that in slots can be associated with a plan element instance by the CAMPS architecture implementors. For example, a resource availability time-line is associated with each resource pool. While this time-line structure is not, strictly speaking, a slot structure, it is maintained in some ways like a regular slot, including being referred to (by certain functions within CAMPS) as if it were one. The presence of these so-called non-slot instance variables is inherited in the same way slots are, but instance variables are considered under the architecture's control, and are neither directly manipulated by the user nor described by the application's knowledge engineer. Since these instance variables are not slots, the user is unable to change them arbitrarily. Furthermore, unlike slots, the values in these non-slot instance variables may change as a result of normal-mode constraint checking. For example, checking that a planned resource utilization is "feasible" may result in updating the resource availability time-line, either to add the utilization when it is feasible, or to remove it if it has been changed to become infeasible.

4.3 Constraints, Rules and Predicates

The CAMPS architecture takes a knowledge-based approach which views scheduling and resource allocation primarily as a constraint satisfaction problem. The facts of some application are expressed (in part) by explicit constraint declarations. The constraints themselves test local conditions.

In CAMPS, a constraint is a conditional statement of the form "if A and B and ...are all true, then the plan element is in trouble if condition D is true." A constraint can be "checked" when all the arguments to A, B, ..., D are known. A checkable constraint is violated if A, B, ...are all believed and D is not strongly disbelieved [7].

Ideally, a plan should not have any violated constraints. However, it is naive to believe that planning problems have solutions in which all constraints are satisfied. The CAMPS architecture uses a numeric measure that reflects the degree of belief in a constraint being violated.

In addition to degree of belief, CAMPS also associates a consequence category with every constraint. The consequence category provides a qualitative measure of the seriousness of the constraint's violation. The consequence category helps to order the constraints for evaluation and provides a metric by which problem-solving strategies can selectively check and relax constraints.

Figure 4 displays the declarative definition of a constraint, and illustrates some of the important features of a constraint specified by the developer.

```
(defconstraint ORIGIN-DIFFERENT-FROM-DESTINATION move
   (origin destination)
   :prohibit (*equals* ?origin ?destination)
   :consequence-category :feasible
   :endorsement 0.3
   :belief 1.0
   :documentation ''Origin of a move must be different from
                    its destination.'')
```

Figure 4: **An example of a declarative constraint definition.**

The qualitative categories of this classification [7] are:

- **Feasibility**: laws of physics or human nature are being violated.
- **Survivability**: normally reusable resources will be consumed.
- **Success**: Implicit goal of a task will not be accomplished.
- **Efficiency**: resources or opportunities are being wasted.
- **Assumption**: a reasonable expectation is unfulfilled.

Constraints can be easily defined. Each constraint declaration creates a data structure with the following attributes:

- **Plan Element.** The plan element that is the focus of the constraint.
- **Involved Slots.** The slots for which the relationship is enforced.
- **Conditions.** This is a list of predicates, some of which may be negated. The constraint is applicable only if the conditions are satisfied.
- **Relationship (Predicate).** A constraint defines a relationship between slots of a plan element. This relationship is expressed as a single, possibly negated predicate that defines the constraint [26].

CAMPS supports three types of predicates: *hand-coded predicates, relation-based predicates*, and *rule-based predicates* (see [5] [6]). CAMPS predicates are different from LISP (or other programming language) predicates in several respects. The standard interpretation—called *normal-mode*—of a predicate is simply to find out if the relationship holds or not. All predicates return three values: (1) `*TRUE`, `*FALSE`, or `NIL` indicating "don't know;" (2) belief; (3) disbelief. [7]

CAMPS predicates can also be evaluated in certain non-standard evaluation contexts. One, called the *make-context*, produces a list of potential changes in plans that might result in the predicate becomming true ("make-true") or false ("make-false"). Another, called the *bias-context*, tries to unify and restrict the arguments to the predicate so that the candidates are likely to satisfy the predicate ("bias-true") or are likely to make the predicate false ("bias-false").

There are three ways to define a predicate in the CAMPS architecture. The first is to write some LISP code. A small number of predicates—typically encoding the meaning of predicates like "PLUS" in an expression (PLUS ?A ?B 10) to claim that $A + B = 10$—have been built this way, but this is clearly not a "declarative representation of knowledge."

A second way to define a predicate is to use the relational database's relations. As an example, the predicate `has-runway-length` in an expression `(has-runway-length ?airbase ?runway ?required-length)` could be defined on the database relation `airbase-facility` using the airbase, runway-designation, and runway-length attributes. This technique for defining predicates simply tells the CAMPS inference mechanisms where to go to look for the answer. The form of the computation is fixed; a declaration provides a mapping of the name of the predicate to the relational database.

The third way to define a predicate is to use a rule. Our formalism uses a single formula to express two closely related logical expressions (see [5] for more details on predicates).

4.4 Variables: Their Structure and Function

In CAMPS, planning is accomplished by filling slots in plan-elements. This is normally accompanied by the firing of constraints which evaluate the acceptability of the slot filler. At a higher level, there are strategies that guide the system in selecting which slots to fill first, how to fill them, and what to do if problems are encountered. This report provides a detailed description of the lower level functions that control how CAMPS produces candidates for a slot and how it attempts to select an acceptable slot filler from the available candidates.

4.4.1 Preliminaries

Each slot in CAMPS is represented by a *variable*. Ultimately, the most important information associated with a variable is its current fixed value and the constraints that evaluate the acceptability of this value. A prerequisite for slot filling is the ability to find candidates for a slot that agree with the specified content for that slot (i.e., they must pass a "type check"). A prerequisite for efficient planning is the ability to select a candidate that will then pass all applicable constraints when tested in the current context. If a serious constraint violation is detected, then it is necessary to change a previously selected value, to decide that the violated constraint does not really apply because of special considerations, or to continue with a flawed plan. However, CAMPS has low level mechanisms that tend to lead to the selection of good values to fill slots. This is accomplished by the use of CAMPS—*generators* that are associated with slot variables.

4.4.2 CAMPS Generators

The purpose of a generator is to produce, upon request, an acceptable candidate to fill a slot. Again, the obvious first requirement is that a generator must know about potential candidates for its slot. CAMPS has different types of generators to support different methods of obtaining candidates. All slot generators are created by `initialize-variable-generator` which uses mainly the content information found on the slot's variable to decide on the type of generator and the source of its possible candidates. The generators discussed below are all built on `camps-generator` and are all used to generate candidates to fill slots in a plan-element. They should not be confused with other types of generators that are used within CAMPS for different purposes.

The simplest slot variable generator is a *list-generator* which, as the name implies, obtains its candidates from a list of internal CAMPS values. As an example, consider an `aircraft-capability` slot which is intended to designate the type of aircraft used in an OCA mission. The slot variable provides the information that the slot content should be an `OCA-ac` capability. It is easy to find a list of all aircraft types that include the `OCA-ac` capability. This list will make up the initial *possibilities* of the list-generator.

Another generator is the *range-generator* which produces candidates that are numbers found in a CAMPS range (made up of intervals which designate start and end values). Representing the possibilities as a range is obviously more practical than trying to provide a (possibly infinitely) long list of numbers. Variables for slots that have content `:number` or `:time` are initialized with *range-generators*. Currently, `:time` produces a generator of integers while `:number` produces a generator of all reals. While there is some need for content types of `:integer`, `:positive-integer`, etc., we have chosen to limit the number of types and to use other means to support variations on `:number`.

The *relation-generator* is yet another type of generator. It obtains its candidates from the relational database associated with a CAMPS domain-specific application (such as AMPS). Consider a slot that has a content of `friendly-airbase`, meaning that it is to be filled with an instance of an `essential-plan-element` possessing the `friendly-airbase` capability. The capability provides pointers to where in the database its possibilities can be found. Airbases might be found in some attribute position of a specified relation. Additional selection information might further specify that we are interested only in those that have "friendly" in some other attribute position. Using this information, all of the friendly airbases can be obtained from the database, and these make up the initial possibilities of the relation-generator. An important difference, however, is that these possibilities are external database values, not internal CAMPS values. When it is asked for a candidate, the relation-generator must select one and then "instantiate" it into an internal value which it returns. Advantages of this type of generator include the natural use of an existing database while limiting the instantiation of database values to those that are actually selected.

There are also various *camps-creation-generators* that produce candidates for some slots. This class of generator tends to create a new candidate rather than selecting one from some known group of existing candidates. As an example, consider a slot that holds a reservation for a resource usage. If there is an existing reservation for this slot, it should already be filling the slot. If there is none, then each existing reservation (in working memory or in the database) should belong to some other slot representing a different reservation. Therefore, a normal action for a generator on such a slot is to create a new instance of a (mostly unplanned) reservation. This type of generator is less relevant to the current discussion which is largely concerned with techniques of selecting a good candidate from a large number of available possibilities.

4.4.3 Selecting Candidates based on Restrictions

Given that a slot has a generator that can produce candidates to fill it, how does the generator decide which of the possibilities should be selected as the next candidate? The default would be simply to select the first possibility that is encountered. This value would become the fixed value of the slot, and then applicable constraints would be fired to test its acceptability. If the value is rejected by constraints, then another value would be tried. In an ideal situation where all possibilities are likely to succeed, this would probably be the most efficient way of selecting candidates, avoiding the overhead of other methods. Of course, it actually has many limitations.

Suppose a range-generator has for its possibilities all integers increasing from 1, and suppose that there is a constraint specifying a violation if the value is less than 1000. Clearly, we want a more efficient alternative to selecting a value, firing constraints, rejecting the value, and then repeating the process until success. The case can be even worse for a relation-generator. While finite, the number of possibilities can still be enormous. And, each possibility is an external value that typically requires a relatively expensive instantiation into an internal value that can be used as a slot filler. List-generators probably would cause the least problem, since they typically have a limited number of possibilities, all of which are internal values. Still, it would be nice to avoid firing constraints on several possibilities, particularly when an expensive constraint is present.

Actually, one step was left out in the algorithm described above. Normally, filling a slot is preceded by firing constraints on the empty slot in the BIAS-TRUE (success motivating) mode. In any evaluation mode, firing a constraint in a situation where all involved slots are fixed will obtain the same answer. A constraint enforcing the *PLUS* predicate on three slots will complain if the context is $6 = 5 + 2$. In the BIAS-TRUE mode, the same constraint fired on $6 = 5 + \texttt{slot-X}$ would send a restriction message to the generator on `slot-X` telling it that a particular constraint wants its candidate to be 1. With that condition, the constraint could then report that it is satisfied. When `slot-X's` generator is asked for a candidate, the existence of the constraint-imposed restriction would lead it to return 1 as its candidate. If it is then used to fill `slot-X`, constraint firing could then be expected to succeed.

Naturally, this is an over-simplification. Some other constraint might simultaneously be imposing a restriction on `slot-X` telling it that the value should be greater than 5. Yet another might want even integers. Generators must have a means of representing all possible restrictions and combining their effects appropriately to determine what candidate should be selected.

One problem is how to determine whether a candidate is acceptable. CAMPS uses the idea of a belief threshold. If the absolute difference between a belief and a disbelief is at least this threshold, then these beliefs earn an overall `*true` or `*false` rating; otherwise they are considered to be uncertain. The current default threshold in CAMPS is 0.6, meaning that `(0.6 0.0)` and `(0.2 0.8)` are examples of minimally `*true` and `*false` beliefs, respectively. If a candidate scores `*true`, then a generator will stop its search and return that possibility. Otherwise, it seems worth the effort to continue searching for a better candidate. Unfortunately, the belief threshold approach does not work very well at the generator level. Overall beliefs in candidates have typically been greatly reduced by evidence combination, particularly by a constraint's consequence-category and by the typically low endorsement weight. Therefore, a filter-tree maintains a best possible score, calculated by obtaining from each leaf filter node the maximum belief that it can return for any candidate, and then combining it with the fixed weights in the tree. Any candidate that matches this score should, therefore, be automatically accepted, since nothing better will ever be found.

It is still the case, unfortunately, that a generator frequently fails to find any candidate that meets acceptability criteria. Each candidate that is considered and found to be substandard is temporarily "suspended" by moving it to a list of suspensions. If the generator runs out of candidates, it will then obtain the highest scoring candidate from among the suspended values. Two candidate beliefs are compared in a manner that gives greater (negative) emphasis to disbeliefs. A `(0.3 0.0)` belief in a candidate, for example, is probably better than a `(0.6 0.2)` belief. If a substandard score is mainly the result of weak weights, then the candidate returned by the generator will probably pass constraints. If the best candidate is actually a bad choice, then constraints will fail. The generator has still produced the best candidate according to the posted restrictions. Although it seems like a lot of work, finding the best of the suspensions has the advantage that it is one of the ways of producing a "best" value. In practice, we have found that relative belief differences within a generator are more important than any absolute beliefs.

When a candidate provided by a generator fails constraints, this means that either there is no acceptable candidate in the current context, or there is a key constraint which acts on the filled slot, but previously failed to post a restriction. This could be because it was not previously fired or because the restriction was too expensive or impractical to calculate in the context that included the empty slot.

Earlier, we examined how slot filling might work without a filter-tree. An unrated candidate would be selected as a slot filler, constraints would fire, and the process would continue until a good candidate was found. At this point, the main change is that the process has been moved to a lower level with the substitution of filter-tree evaluations for constraint firing. This is in fact a very significant improvement because instead of testing each candidate against all of the applicable constraints at once (and invoking all of the associated support functions for doing this), the candidates are tested against individual constraints and filtered separately. Repetitive application of constraint testing functions are minimized. However, it leaves us with an analogous trial and error search for a good candidate. For the typical list-generator, this technique is acceptable. For the other two generator types, there will still be occasions when good candidates will be found only after huge numbers of trials. This is undesirable even allowing for the large speed increase for each candidate test. A solution to this problem is the use of "prefiltering" in addition to filtering.

4.4.4 Prefiltering Generator Candidates

Consider an example where a range-generator has initial integer candidates `((0 +infinity))`. Suppose two conjunctive restrictions have been posted, the first favoring values >700 and the other wanting values <2500. Using standard filtering, we might find a good candidate only after 700 tries. However, it is easy to see, and also easy to represent, the fact that values in the range `((701 2499))` are somehow favored over those in `((0 700) (2500 +infinity))`. We could suspend the substandard values as a group and leave the favored values on the possibilities. Having done this, the generator would then consider 700 as its first candidate. This is the basic motivation for prefiltering.

Slot restriction messages are sent out during *hand-coded* and *dbpredicate* evaluations without any consideration of the type of generator on the slot to be restricted. It is up to the generator to decide how to implement the restriction, or even whether or not it can implement it. The same restrictions sent to different types of generators will normally result in different filters in the leaf nodes (FIFTN) of the filter-tree (see Appendix A for more details). All of these filters must possess certain capabilities. Each must be capable of taking an internal CAMPS value and returning beliefs in its acceptability. Additionally, each must be able to return, upon request, several other items: the maximum belief that it will give to any theoretical candidate; a string documenting its filtering actions; the restriction message and arguments that led to its restrictions; and whether or not some other filter enforces the same restriction that it does. In addition, some filters have the capability of prefiltering candidates. This capability appears in their ability to return a representation of their candidates weighted by belief in acceptability.

Prefiltering is ideal for range-generators because of their ability to represent large numbers of candidates in a compact manner. Prefiltering is still relatively expensive, however. Each leaf filter that supports prefiltering, returns a weighted-range. This associates beliefs with various subranges of the candidate range. Usually, this is a fixed belief for a given subrange. The other option currently supported is to have two beliefs associated with a subrange consisting of a single interval. The two belief pairs are for the values at the start and end of the interval, while other candidate beliefs vary linearly between the end values. Once a weighted-range is obtained, it has to work its way up the filter-tree, experiencing the same types of evidence combination that a single candidate would encounter. The final result is a weighted-range showing all possible candidates ranked by the combined belief of all restrictions. Subranges with substandard beliefs can be immediately suspended. Supplying a candidate now consists of getting the first (and best) value from the possibilities range or else the best of the suspensions. The generator still tests this individual value against the filter-tree, but it should be found to be either acceptable or at least the best substandard value.

List-generators can also support prefiltering. Since there is usually less to gain, the generator may not engage in prefiltering for all restrictions. A filter that supports prefiltering returns a weighted- list of candidates. These are simply lists of internal values with an associated belief for each value in the list. There may also be a NIL list with a belief for all values not found in the union of the other lists. At the simplest (and least desirable) level, a list-generator filter could create a weighted-list for prefiltering by individually testing each of the possibilities. A restriction imposed by an *EQUALS* predicate provides an example where prefiltering is more compatible. It would produce a weighted-list providing one belief for a specified candidate and another belief for "all other" candidates. This could be done without even looking at the possibilities, except to check that the specified candidate is in fact a known possibility.

Prefiltering also greatly improves performance of relation-generators. Our original approach was to perform normal database operations on relations involved in different restrictions of the same slot. The acceptable possibilities would then be selected from the result. This approach had two problems. First, some situations led to some very expensive relational join operations. Also, all restrictions based on the database could be made only to relation-generators, which were not compatible with other generators. All restrictions were considered to have absolute belief (if present in a relation) and absolute disbelief (if absent).

Now, a relation-generator starts out with possibilities from a relation. A database restriction is posted as a filter that tests a candidate during filtering for its presence in a certain relation (first converting the candidate to an internal value). It returns different beliefs based on the presence or absence of the candidate. Prefiltering is supported using weighted-lists as in list-generators, but with external values. Thus, the current treatment of database constraints is identical to what is the case when the constraints are not based on *dbpredicates*.

Any generator can have a database restriction which will be combined in the normal manner with other restrictions. Similarly, non-database restrictions can also be posted in a relation-generator's filter-tree, evaluating an internal candidate in the normal manner. Although infrequent, a range-generator could, for example, have a restriction that a candidate should receive one belief if found in some location of a specified relation and another belief if not found. This restriction could even be converted into a prefilter weighted-range if this operation were deemed to be relatively inexpensive. Similarly, a RELATION-GENERATOR could have a restriction that is not based on the relational database. Consider a restriction that the filler of a slot must be a capability possessed by some plan-element. During filtering, the filter will test whether that plan-element has the candidate as a capability. Since a plan-element has only a small number of capabilities, it is also easy to support prefiltering. This involves an inexpensive conversion of these possibilities into external values, all of which receive one belief pair while all other values are assigned the "not present" belief.

4.4.5 Applicability and Efficiency Considerations

When a candidate is tested and then relegated to the suspensions, it is also marked with the filter-tree's current timestamp. This timestamp indicates the time of the last change experienced by the filter-tree. Whenever a change occurs (adding a new filter, or removing or modifying an existing filter) the filter-tree receives a later timestamp. Any suspended value, with a timestamp that predates the filter-tree's, might now receive an acceptable or better rating. A change does not necessarily cause an immediate review of the suspended values. In this case, the generator continues to test candidates still on the possibilities. But if it runs out of them, it will check for outdated suspensions. If no acceptable candidate is found, it then installs the best suspended value as its candidate.

Suppose a constraint evaluation posts a restriction on a slot. Then a short time later, the constraint is fired again without any changes having been made that would affect the original restriction. We do not want to do a lot of unnecessary work and end up with two separate but identical restrictions posted in the filter-tree. One way to avoid this, of course, would be to not fire a constraint when no change has occurred that would affect the result and side effects of the constraint's previous firing. In practice, it can be difficult to recognize that this situation exists. But refiring the constraint is actually fairly efficient, anyhow. A restriction is installed at the end of a path that exactly reflects the source of the restriction. When a restriction is received, the installation code follows the existing filter-tree paths, diverging only if the new path does not yet exist. If nothing has changed, then this process will reach an existing filter node and will also recognize that the existing filter already is imposing the desired restriction. The filter-tree would not be changed and the new filter would simply be discarded. (The new filter is initially created in skeletal form. Most of the work in filters, such as creating a weighted-range, is delayed until triggered by an actual request.)

An existing filter may also be modified. Suppose `slot-X` is restricted to be greater than the fixed value of `slot-Y` which subsequently receives a new value. In this case, installation of the new filter again reaches an existing filter node. This time, however, the existing filter is found to be different and is replaced by the new restriction. Although the structure of the tree has not actually changed, its evaluation standards have changed and the filter-tree is marked with a new timestamp.

There is a problem, however, when a restriction posted by one firing of a constraint is entirely absent in a subsequent firing. Perhaps it was imposed by the antecedent of a rule which, due to changes, is now considered to be inapplicable. Or, perhaps the constraint no longer passes the constraint conditions. In this case, a restriction already posted in the filter-tree will be inapplicable. Recognizing that it needs to be removed seems difficult because this restriction will not be encountered during the new evaluation. It turns out that this is all that we need to know. During a constraint evaluation, the filter nodes (FIFTN) of all encountered restrictions are collected and then stored on the constraint instance. On the next firing, a new list is collected. Any node present in the older list but not present in the new list represents a restriction that no longer applies. Remember that a repeated or modified restriction always ends up in the original FIFTN, so comparing the new and old lists works correctly. Having the FIFTN also makes it simple to unwind the restriction. A restriction node is removed from its parent's subnodes slot (we currently shift such nodes to an oldnodes slot, on the assumption that they are likely to be used again during a subsequent evaluation). If the parent node finds it no longer has any subnode, then it similarly removes itself from its parent, etc.

Another case requiring restriction unwinding occurs when a constraint evaluation posts a restriction and then later during the same evaluation discovers that the restriction is not applicable. This is most apparent during rule evaluation. Each rule antecedent is to some extent a condition for all other antecedents. Suppose several antecedents are evaluated. If exactly one restriction is posted and all antecedents are true, then that restriction is applicable. However, suppose after making a restriction, a later antecedent is false. Or, suppose a later antecedent requires a restriction on another slot in order to be true. In either case, the rule is considered to be inapplicable, and the restriction needs to be unwound. Fortunately, the rule evaluation is already collecting its filter nodes, as discussed above. So it knows what restrictions have been imposed by its antecedent paths and is able to unwind them easily. We have also looked at the possibility of a restriction in a disjunctive path being rendered inapplicable by the result of another path. For example, the first path might need a restriction to satisfy a predicate, but then the second path might support the predicate without restriction. The restriction may seem unnecessary, but we currently leave it in force. If the restricted path is capable of complaining, then failure to satisfy the restriction could cause a constraint violation despite the success of the other path. If the restricted path only has endorsement capability, then the restriction is probably unnecessary. On the other hand, if everything else is equal, why not select a candidate that this restriction likes? If there are good reasons for selecting some other candidate, then this restriction will not oppose such a choice.

The use of prefiltering can be controlled by the user. The normal (and also most extreme) practice is to prefilter each time that a filter-tree is changed. Empirically, this seems to provide the best results. Prefiltering can also be turned off. Filters that support prefiltering also support normal filtering anyhow, so the system should run. At times, running without prefiltering can lead to serious efficiency problems. CAMPS generators are capable of catching some of these problems by detecting situations where a request for a possibility leads to an excessive number of failures. In one configuration, a generator can be told not to prefilter automatically, but to invoke prefiltering only after filtering has rejected a certain number of possibilities during a request for a candidate. In other words, it avoids the expense of prefiltering if good candidates are easy to find. Finally, a generator can interrupt searches for a good candidate after an excessive number of values are considered unsuccessfully. Currently, this allows only the user to intervene, but we could perhaps alternatively call on higher level CAMPS procedures to provide guidance.

4.5 Problem Solving Strategies and Metaplans

In CAMPS, strategies can be provided that provide top-down intelligence to guide planning. Constraints provide sufficient bottom-up knowledge to support the automatic filling of a single slot with the best candidate. The metaplanning mechanism supports the automatic planning of the more complex domain tasks, consisting of large sets of slots associated with different tasks and sub-tasks. Metaplans express control knowledge as an ordered sequence of planning steps. Some metaplans have specific knowledge of how best to plan a particular task, including what slots to fill in what order. Variations on these metaplans might be particularly applicable for replanning and others might be specialized for replanning in time critical situations. Other metaplans are designed to fix problems (e.g., a constraint violation) using knowledge ranging from general planning considerations to very domain specific techniques for fixing specific problems.

The strategic component of CAMPS is driven by the posting of goals. Since it is possible that many metaplans might advertise their ability to achieve a particular goal, other metaplans are designed to select the most promising ones to execute first. This decision can be based on which is the most specialized, whether the planning context satisfies applicability filters of some strategies, whether certain global goals have been posted (e.g., conserve certain resources, maximize safety), etc. Metaplans have potential for evaluating the usefulness of a plan and also to explain what planning decisions were made and why.

4.5.1 Metaplans

Metaplans are mostly declarative descriptions of problem-solving strategies containing three kinds of information:

- Goal Pattern giving the goal symbol the metaplan matches, the required goal arguments, the optional goal arguments, and absolute applicability tests based on the arguments. The applicability tests themselves may be stated in terms of the names of LISP functions that are free of side-effects.
- Applicability measures quantify in absolute terms how well the given metaplan is likely to work in achieving the specific metagoal along a number of different dimensions (e.g., specificity of the metaplan, speed of solution, expected solution quality).
- Template for creating an active problem-solving agent to execute the metaplan. The template names the standard control flow of the active agent and a list of alternating keyword and arguments, where the keywords are more or less specific to the given standard control flow. Some of the parameters may be names of procedures or lists of names of procedures.

The inclusion of functional parameters within the description of a metaplan implies that metaplans are not purely declarative. We however claim that the metaplan formalism is mostly declarative because the procedural portions (the functions named in the metaplan description) are severely constrained to be free of side-effects, which prevents encoding arbitrary actions into them.

4.5.2 Metagoals

We separate the goals of a plan (which might be to interfere with an enemy's ability to wage war) from the goals of a planner (metagoals). These metagoals concern the planning process itself, for example:

1. Parameterize the way in which tasks in a project will be performed.
2. Find a "rough cut" schedule to identify critical resources.
3. Assign resources to tasks taking suitability and availability into account.
4. Schedule tasks.

Syntactically, a metagoal is simply a goal symbol with some number of parameters. Metagoals are formed automatically by CAMPS. Given a metagoal, CAMPS finds all the metaplans that are applicable by pattern matching. If there are several matches, then the best is selected according to heuristic comparisons based on the applicability measures. Finally, the "best" metaplan's template is used to create an active problem-solving agent, with the less applicable metaplans held in reserve in case the "winner" fails [7]. In keeping with the CAMPS philosophy that as much knowledge as possible should be encoded declaratively, the basic agendas from which domain-specific metaplans are constructed are contained in an agenda hierarchy.

4.5.3 Viewing Plans as Trees of Tasks with Associated Resources

The CAMPS architecture uses a suite of AI techniques to represent knowledge about a particular planning application. These techniques include notions about specialization and generalization, constraint-based heuristic search, and evidential reasoning. (See [7] for more details).

A plan—the solution to a planning problem—will consist of some number of top-level tasks, some of which may be interconnected by precedence relationships. Each of those tasks may have one or more subtasks which may themselves be interconnected by precedence relationships and have subtasks. That is, the tasks of a plan are arranged into task/subtask tree structures, with the top-level tasks acting as the roots. Furthermore, each task has a number of slots that further parameterize the task and the resources it needs. For example, the planned start and finish times for a task each fill a slot. A complete plan is then the full collection of tasks and subtasks with all their slots appropriately filled.

CAMPS uses a hierarchical approach to planning that can take a plan complete at one level of detail, and further refine and modify that partial solution so that it is complete at a greater level of detail. Detail in a plan refers both to the number of slots filled in its tasks (i.e., a less detailed plan has fewer slots filled in its tasks than does a more detailed plan), and to the depth of the task/subtask trees.

The CAMPS architecture views a plan as a set of projects. [10] A project is a collection of tasks arranged into a task/subtask tree. CAMPS identifies a project with a tree of tasks, the root task of which is used to represent the entire project. The view CAMPS thus takes is somewhat unusual:

1. A plan may consist of several disjoint task/subtask trees. Alternatively, all the tasks of a plan may be in a single tree.
2. A project need not be a top-level task; a task viewed as a project may itself be a subtask of a higher-level task.[7]

The generality of being able to view any task in the task/subtask tree as a project in its own right is necessary: in top-down planning, leaves of the task/subtask tree become lower-level projects; in bottom-up planning projects are assembled as tasks within higher-level projects.

The project-planning strategy is described by a metaplan as a sequence of achieving some number of milestones for all the tasks in a project.

Milestones are usually testable situations that should arise after certain sets of tasks have been performed. Metaplanning milestones are situations that arise during the plan formulation process.

In CAMPS the specific milestones (see [7] for further information), arranged in their expected order of achievement, are:

- *Task Template Expansion.* Necessary subtasks and prerequisites of task are described in CAMPS by templates and stored as part of the plan element declarations. Much of the task/subtask tree and task precedence relationships can be automatically derived by an inference mechanism called template expansion.
- *Resource Template Expansion.* Some templates describe incidental and abstract resource needs. By expanding these templates, most of the resource needs of tasks can be identified.

[10]Project planning is a special instance of the general planning problem in that a complete list of tasks and subtasks can be enumerated from the problem specification. In long-term planning, the initial problem specification is usually incomplete, thus the tasks and subtasks can only be partially enumerated. Project planning emphasizes resource allocation and scheduling, rather than discovering the sequence of steps that accomplishes some goal. Military mission planning, as well as large-scale construction, political campaigns, and complex surgery all fit within the project planning framework expected by the CAMPS architecture.[5, 7]

- *Preliminary Task Parameterization.* Some of a task's parameters are needed in order to select appropriate types and quantities of resources. This milestone is passed when the slots for those parameters have been filled.
- *Find Temporal Bounds.* Given the precedence relations and task/subtask tree structure, upper and lower bounds can be found for the start and finish times for each task in the tree.
- *Estimate Resource Allocations.* Assignment of resources to tasks proceeds in two steps: the first step uses approximate times for the beginning and completion of the usage to reserve a resource. Overbooking of resources may be allowed. The purpose of this step is to discover which resources are in short supply and when those shortages occur.
- *Schedule Task.* A task is scheduled when specific times are derived for the task's start and finish.
- *Reserve Resources.* Given a resource reservation, an allocation of the resource can be made to a task for a specific time period. To the best of the system's knowledge, allocations are not overbooked.
- *Final Task Parameterization.* When resources have been assigned and a task has been scheduled, there may be some final parameterization required.

The rationale behind breaking the project planning process into a sequence of milestones is to allow for the substitution of global strategies for local ones at milestone boundaries. Such a global strategy will usually use its own technique for enumerating tasks and slots within those tasks to be filled.

A good example of a global strategy is one that, for tasks using several scarce resources, finds both a schedule and a set of assignments (see [32] for a comparison of "state of the art" exact algorithms for solving this kind of problem). Consider Stinson's [37] as a specific example; such a strategy would (partially) replace local strategies for three milestones: estimate resource allocations, schedule task, and reserve resources. Since we have no way of knowing which of those milestones will occur first in a project-planning strategy, we need to make sure that all three milestones are recognized as metagoal symbols signaling that Stinson's algorithm should be used.

Many global strategies for resource allocation first find an acceptable solution (e.g., schedule with assignments) and then try to improve the solution. Local strategies also use guess and back up while filling slots (see [7] for details).

4.5.4 Plan Repair using Delta-tuples

Constraints are used in the CAMPS knowledge-based planning system to represent those propositions that must be true for a plan to be acceptable. One mode of interpreting a constraint determines its logical value. A second mode inverts a constraint to restrict the values of some set of planning variables. CAMPS introduces a third mode: the "make" mode. Given an unsatisfied constraint, make evaluation mode suggests planning actions which, if taken, would result in a modified plan in which the constraint in question may be satisfied.

These suggested planning actions—termed *delta-tuples*—are one source of raw material for intelligent plan repair. They were intended to be used both in "debugging" an almost right plan and in replanning due to changing situations. Given a defective plan in which some set of constraints are violated, a problem-solving strategy would select one or more constraints as a focus of attention. These selected constraints are evaluated in the *make* mode to produce delta-tuples. The problem-solving strategy would then review the delta-tuples according to its application and problem-specific criteria to find the most acceptable change in terms of success likelihood and plan disruption. Finally, the problem-solving strategy makes the suggested alteration to the plan and then rechecks constraints to find any unexpected consequences. While delta tuples were implemented in CAMPS, experience with the system revealed that they were less useful than was originally thought, and they are not actively used in the current system. For this reason we do not provide a detailed examination of delta-tuples in this report. For a full description of delta-tuple usage see [26] and [7].

4.6 Scheduling

Operations research shows how, in various special cases, to simultaneously achieve several milestones over a task tree in a global manner. Generally speaking, no real planning problem precisely fits the special case required by an OR algorithm, so one is generally content to ignore some considerations, find a partial solution, and then fill in the rest in an ad hoc manner, making changes in the OR solution as required.

The duration of some kinds of tasks (e.g., building a house) is a function of how much of several types of resource (e.g., carpenters and masons) is applied to the task. Both resources and overall tardiness have associated costs, so we would naturally like to find the overall minimum cost way to allocate resources. The well-known critical path method in management science uses linear programming to find such a global minimum.

During 1988 we undertook an experiment to build a new scheduler for AMPS which uses an application of branch-and-bound search for an optimal schedule, using Stinson's selection and pruning heuristics (see [7] for details). The goal of the experiment was to develop a scheduler that would use a global scheduling strategy for "enumeration" of an optimal schedule, taking this ideal schedule as the starting point in producing a feasible schedule which satisfies resource constraints. With this approach, a global planning view finds the best starting point, local adjustments complete the work.

Essentially, this experimental scheduler is a metaplan that follows two basic steps: devising an optimal schedule, and committing to some approximation of that schedule. In devising a schedule, it applies branch-and-bound search, implemented as an agenda, while ignoring resource constraints almost entirely. The schedule produced is "ideal" in that it is the best possible and almost certainly unattainable; all tasks are scheduled for earliest start, but careful resource tracking is postponed.

In the resource allocation phase, it attempts to assign task schedule times while tracking resources, succeeding or generating suggestions for changes (delta tuples). An agenda provides the procedural skeleton of branch-and-bound-search. It accepts the relevant parameters—tree initialization, node generation, heuristics for node selection and pruning, and so on—as instance variables, and carries out its search using the supplied functions. A metaplan is used to invoke a domain independent agenda to supply a partial schedule tree representation and Stinson's heuristics.

Search parameters are supplied by metaplans which invoke the agenda, so that varying search strategies (within overall branch-and-bound search), search goals and search space representations can be produced with the same agenda invoked in different environments. The schedule produced through branch-and-bound search is in fact used in two ways: it provides initial values for schedule times, and it guides the scheduling task. Instead of scheduling a task, then its subtasks, and so on, using this experimental schedule will cause CAMPS to try to schedule the tasks which are best started first, then second, and so on. Intuitively, this seems reasonable: if the system encounters problems, they will typically be due to resource restrictions, and there will be only one basic option for resolving the difficulty: postpone one of the tasks involved in the resource conflict.

4.6.1 Description of a Schedule

A schedule could be as simple as assigning a time interval to each task, [11] or as complex as a description of inter-agent communication protocols. Generally speaking, the "richer" the scheduling vocabulary, the more "planning" is done at "plan execution." But by postponing planning decisions, the planning problem generally becomes harder because the plan-time planning needs to take all eventualities into account. Points along the schedule complexity dimension include time intervals, partial precedence graph (with unplanned coordination among parallel tasks), single-process conditional branches, single-process general recursive plans (the statement of the "plan" has the control structure of a general recursive procedure), multi-agent cooperative plans, and multi-agent adversarial plans.

4.6.2 Tasks

In the most general terms, a solution to a planning problem contains the following:[12]

- A set of tasks that are to be performed.

[11]See [6] for a description of how the PERT scheduling technique can be re-interpreted as a problem-solving strategy, and for a description of the hierarchical PERT (HPERT) developed for use in CAMPS.

[12]This idea is described in more detail in [5].

- An assignment of resources to the tasks.
- A schedule describing the sequence in which the tasks should be performed.

Goal states, constraints, and utility functions are parts of the statement of a planning problem (although they may be elaborated on as part of solving the planning problem: the planner may uncover additional constraints while trying to construct the plan).

"Tasks" refer to things that happen during the execution of a plan, not to things that must be done to prepare a plan (in CAMPS the latter are called "metagoals"). The notion of a "task" can also be broadened to include "applying a test" and "performing some operation," and "making a decision." A specific task can be completely described by giving a list of its starting and ending events (often given as times), a set of needs and the resources that are meeting them, and a set of other associated tasks. Any instance of a specific task type can be described using some standard form, with "blanks" on the form being filled in with:

- something indicating when the task starts and stops (i.e., when the task is scheduled)
- specific needs filled with quantities of resources
- specific kinds of support provided by other tasks

In CAMPS, tasks, resources, relationships between tasks and between tasks and resources are described in the following structures: the plan element hierarchy, the relational database, constraints, and the task template mechanism. The plan element hierarchy provides information on the conceptual generalizations and the capabilities of tasks and resources, e.g., a resource is sharable, movable, one of a kind of aircraft. The database provides information on specific tasks and resources. A constraint might be used to specify a certain relationship between tasks, between resources, or between tasks and resources (a task of type X always uses a resource of type Y). Finally, the task template mechanism provides a declarative mechanism for specifying the predecessor/successor and parent/child relations among tasks. With this mechanism, the user or the metaplans can plan to greater or lesser levels of detail (in terms of task/subtask decomposition).

4.6.3 Resources

Resources are organized into pools. A resource pool represents one or more resources. When representing more than one resource, all the resources in the pool are considered interchangable and indistinguishable. Pools themselves may be grouped into pool sets.

A local pool is a pool of resources set up for a specific task so that that task's subtasks, when they require resources, can come to the local pool first, before finding some other pool in the usual way. The idea is to approximate the average usage of some resource (e.g., people) at a high-level task so that the low-level tasks will not need to be explicitly represented in order to calculate resource needs.

Without some ability like this, a planning system is forced to "blow out" all the low-level detail plans, introducing too much too early in the planning process. The local pool concept is therefore what enables CAMPS to plan at multiple levels of detail. A task announces that it would like to have one or more local pools either by setting up slots to specifically hold the local pool, or by using the task template mechanism.

Resource Tracking

The availability of resources is typically the most critical factor in the type of planning supported by CAMPS. While many plans might interact in a very direct manner, competition for a limited number of resources often affects the planning of all tasks in a less direct but very significant manner.

Assigning resources to tasks in a plan addresses problems of availability, cost, and suitability. Evaluating a particular resource's suitability for a task and that task's resultant likelihood of success involves expert knowledge and judgment. The CAMPS architecture represents and uses expert knowledge to interactively solve resource allocation problems.

From the perspective of a planning system, resources fall into three rough classes:

- *Primitive Resources.* are those things whose availability can be known a priori. The numbers of primitive resources may change in predictable ways.
- *Aggregate Resources.* are those resources that are always composed of specific numbers of simpler resources, but are otherwise created dynamically. For example, a "flight ready aircraft" consists of an airplane, fuel, pilot, etc., each of which is itself a resource.
- *Composite Resources.* are those resources that are composed of other resources, but whose composition is not known a priori but must be "invented."

Planning problems involving composite resources are generally harder than similarly formulated problems in which all resources are primitive. The presence of aggregate resources in an application impacts the way in which resources utilizations are handled.

How CAMPS Reasons About Resources

In CAMPS,resources (see [7] for a further description) are used from the start to the finish of tasks. Indeed, a task starts when it begins using resources (like people and floor space). CAMPS uses a notion of partial utilization to represent both intermittent usage and usage of an uncertain duration. With the CAMPS formalism, the only ways to say "Task A lasts for 3 days, and uses a truck on day 2" are: (1) claim the truck is partially used at 33%, or (2) claim that task A has a one-day subtask, and that the subtask uses the truck. The task/subtask semantics of CAMPS allows for a task reserving a resource for use by its subtasks; the two claims about the truck are therefore not mutually exclusive.

CAMPS has the ability to track the availability of various types of resources to different degrees of detail. A quantized-pool supplies one type of resource to consumers. This may be a pool of a single major end-item that is tracked individually, or it can be a pool of many like items (quanities of fuel, boxes of paperclips) that are tracked collectively. In either case, a pool always knows how many resources are available at any given time. The tracking method will depend on the demands of the domain. A single pool might be sufficient to track the availability of all of a company's cars. But if we start distinguishing between the cars (the one the president gets, maintenance records, etc), then individual pools will be required. A pool-set for a type of resource is a pool that has some sort of control over the quantized-pools that actually supply the resource. A company is a pool-set for the company cars (and for many other types of resources) that are tracked individually or collectively. Requests to the pool-set for a particular resource will be passed on to an appropriate quantized-pool under its control. The pool-set can also answer general questions about the resources that it indirectly supplies (e.g., the collective availability of end-items from separate pools).

Some resources are aggregate resources, meaning that they consist of a collection of various quantities of different resources. These are tracked by aggregate-pools which serve as quantized-pools for the aggregate resource. But they must first obtain the components of the aggregate from appropriate quantized-pools that supply required items. Resource-utilizations represent a contract between a supplier (pool) and a consumer (task) for a quantity of a particular resource for a certain time interval. Consumable resources differ only by having an expected return that might be eternity. All pools and utilizations are instances of plan-elements whose slots are filled during the planning process. This means that tasks and resources-utilizations are both plan-elements and are manipulated in similar ways by CAMPS. This supports a dual view of planning, either as a sequence of tasks that consume resources, or as a series of resource utilizations that enable certain tasks. The ability to take the latter view is beneficial when the system is concerned with resource problems.

4.7 Hypothetical Worlds

CAMPS supports the ability to hypotheically explore different planning alternatives. It uses a coarse, assumption-based approach. At any time, the values seen by the system are consistent with the current world, which consists of a collection of assumptions. These assumptions are obtained by selection from previously defined sets of assumptions. Since the assumptions in a set are mutually exclusive, no more than one assumption from each set is selected to create a given world. Assumptions are atomic in nature. It is assumed that the planner will take steps to make the current world "agree" with its assumptions, but there is no way of automatically recognizing or enforcing this. Hypothesizing is a very explicit action. Normal slot filling or modification is considered as a desired change to the current world with no need to retain the previous state. The planner must explicitly create assumptions, build worlds, and then switch to a world in which the next change should be valid, and out of a world that should retain the previous state Hypothesizing can be used to explore alternative plans by both the human user and the metaplanning system. An additional use might be the creation of contingency plans. It is possible to create various alternatives for each of several plans. Once the planning is done, the system monitors external changes. As these changes match some contingencies, it is possible to quickly select a new world consisting of the old assumptions that are still valid and the contingency assumptions that have just become true. This world would, ideally, once again consist solely of valid plans.

4.7.1 Multiple Hypothetical Worlds

Having multiple hypothetical worlds is useful in a planning system for two distinct but related reasons:

1. When planning using a generate and test paradigm, each choice could spawn a hypothetical world. The multiple worlds can then be easily compared and switched among.
2. The user of the planning system may want to explore various assumptions, options, and alternatives.

The difference between these two uses of hypothetical worlds is in the assumptive granularity required. At one fine-grained extreme, every distinct choice for each decision variable of the plan is treated as an independent assumption. At the other coarse-grained extreme, the connection of world to distinct choices is lost, so that changing the value of one or more decision variable does not necessarily introduce another world.

Naturally, one would like to use a very fine assumptive granularity. However, in a typical planning application of the CAMPS architecture, there will be hundreds of plan elements, each with a dozen or so distinct decision variables, giving us thousands of decision variables, many of which are continuous. A fine assumptive granularity may well involve hundreds of thousands of apparently independent assumptions.

4.7.2 Hypothetical Worlds and Comparison with ATMS

CAMPS uses a scheme similar to [16] to tag the hypothetical world in which a slot is filled with a particular value. While we wanted to use something like the ATMS scheme developed by deKleer, this scheme, as described in the literature, necessarily has a fine assumptive granularity which has the potential to result in hundreds of thousands of assumptions.

The reason we consider ATMS to be necessarily fine-grained revolves around the way assumptions representing choices are discovered to be incompatible. Suppose we have a decision variable V, and under some set of assumptions A we compute V to have value 1, while under the same assumptions A we also find that V has value 2. In the ATMS scheme we then conclude that some of the assumptions in A are mutually incompatible. ATMS records this set as being in the no-good set, and it becomes impossible to discuss the value of V under assumption A. The ATMS scheme centers on how this no-good set is maintained and used.

A coarse assumptive granularity would allow the above situation to be interpreted differently. Under assumptions A we compute V to be 1, and later find that V should have the value 2. Rather than indicating A contains mutually incompatible assumptions, this merely indicates that something changed that was below the assumptive granularity. The value for V is simply changed to 2, and the fact that 1 is also a possible value for V under assumptions A is lost.

Under ATMS, if the user decided to change the value of V from 1 to 2, ATMS would consider that the user had changed assumptions: that is, the user was assuming that $V = 1$, but is now assuming $V = 2$. That is, the user has switched hypothetical worlds, so all conclusions that were based on $V = 1$ are part of the previous world, but not the current. The situation for $V = 1$ remains accessible.

One of the advertised advantages of the ATMS scheme is that of fast world-switching. By recording only those assumptions which were actually used in deriving the value for a decision variable, one can avoid recomputing them for a different hypothetical world when the difference does not involve the dependent assumptions. When using a coarser assumptive granularity, one may need to be somewhat more conservative: a computation may depend not only on the assumptions justifying the computation's input, but perhaps some others that are below the granularity.

In summary, when using a coarse assumptive granularity, one can use many fewer assumptions. One can retain the ATMS fast world switching. However, one looses much of ATMS's ability to detect inconsistent assumptions.

When using the assumption mechanism, we are working in a specific world. That is, when we access values, we are interested only in values whose assumptions are consistent with the world. Furthermore, when we deduce new values, we will assign them this world's tag.

Starting with the null world, each SOMEC (Set of Mutually Exclusive Choices) introduces mutually exclusive hypothetical sub-worlds, in the sense that everything that was TRUE in the null world remains true in the sub-world, but there may be slots filled in the subworlds that are not filled in the null world. Similarly, given any world and a SOMEC that is not already included in that world, mutually exclusive sub-worlds of that world can be introduced.

4.7.3 SOMECS

We think of a SOMEC as a set of positive assumptions. That is, we might assume that one of cases A, B, or C holds. Each SOMEC is further augmented by two other implicit assumptions: the splitting assumption corresponding to none of the positive assumptions, and the null assumption corresponding to having not made any of the SOMEC's assumptions. We'll allow the splitting assumption to be named and annotated, but the null assumption is "owned" and controlled by the assumption mechanism. Thus, the minimum SOMEC includes three possibilities: an assumption A, the negation NOT-A as the splitting assumption, and a third corresponding to assuming neither A nor NOT-A.

There are several ways to represent world tags. We represent a tag as a pair of bit vectors, which are termed the mask and the pattern. The NULL tag is represented by all zeros in both the mask and the pattern.

The primary description of a world is simply the user-entered textual description of the world. Since each assumption and each SOMEC also has textual descriptions, the human user has access to multiple levels of detailed explanation of the current and alternative worlds. However, textual descriptions do nothing to help the CAMPS architecture recognize when a world's assumptions have been met or violated.

In order to allow for automatic recognition of worlds, we have augmented the world description so that the architecture can have some insight into what the assumptions and world configurations mean. We have attached a predicate expression to each assumption, so that the assumption is consistent with the current world if and only if the predicate expression evaluates to TRUE in the current world. Since SOMECs introduce branches in the world-lattice, we can examine each SOMEC, decide which (if any) of its assumption's predicates are TRUE in the current world, and then suggest to the user that a more specialized world is consistent with the current state of affairs.

4.7.4 Private Worlds

The contents of the CAMPS architecture working memory is lost at the termination of a planning session. Typically, working memory is stored away into the relational database before a session is terminated. The contents of the database persists. Furthermore, a planning session is assumed to coincide with the work of a single user.

We would like the hypothetical worlds mechanism to enable a user to set up a private set of worlds that other users cannot normally see. We provide this capability by allowing a user to establish private SOMEC that only he sees. This SOMEC is present in working memory only for certain users - specifically for users whose name (or user group or some other identification associated with the user's session) matches a "privacy string." The *SOMEC-definitions* relation contains an attribute holding a privacy-string recording such a situation.

When the database is accessed, if the relation is tagged on a relation per world basis and if the relation version's tag refers to a SOMEC for which the user's session did not match the privacy-string, then that version of the relation will not be accessed. For relations organized on a tag per tuple basis, no tuple will pass an ASSUMPTION-SELECT whose tag includes assumptions that refer to some SOMEC for which the user's session did not match the privacy-string.

For the CAMPS internal representation of tags, no portion of the bit string represents a SOMEC for which the user's session did not match the privacy-string.

4.8 The Relational Database

Plan elements are mapped to the relational database by two different kinds of database declarations attached to their capabilities. These declarations specify:

1. Primary relations containing tuples that directly describe the individuals with this capability (i.e., in this object class). Associated with each primary relation is a primary attribute, which is the designated key in this database relation for this capability.
2. Refinement restrictions which are attributes. For an entity being instantiated, every restricted attribute must have some value in the database that is in the set of values given in the restriction.
3. A declaration of a set of primary relations for each capability in the AKO hierarchy which can be instantiated. These primary relations may be declared on the capability itself, be inherited from its more generalized parents, or the two may be merged. Specifically, the set of primary relations associated with a capability is the union of primary relations associated with its components.

4.8.1 The use of RDBMS in CAMPS

The CAMPS relational database management system (RDBMS) was made available as a prototype early in 1987 (which is documented in [2]). CAMPS uses a relational database to hold information about the tasks it plans, the resources those tasks use, and other incidental information (e.g., predictions of relevant factors, status of facilities, topology, politics, costs). The relational model currently used by CAMPS is implemented by a custom built single-access, in-memory RDBMS.

CAMPS has been designed to use an external RDBMS to:

1. share information among different stations (not implemented)
2. receive information from, and communicate to, non-CAMPS sources
3. maintain intermediate records of the planning process
4. record the planning decisions and justifications (not implemented)
5. maintain alternate versions of a given plan (not implemented)

The advantages provided by a DBMS are well known. The advantages of a relational database management system over other database organizations are primarily uniformity of representation and the high degree of independence between conceptual organization and actual implementation. The following lists the advantages of using a relational DBMS enjoyed by CAMPS:

1. **Reliable and Robust Data Store.** The DBMS provides restart and recovery in case of hardware and software failure and guarantees structural integrity and consistency. A DBMS maintains its own storage and buffer management, including management of interrelations among clusters of data. The DBMS maintains and provides alternate access paths to the same data, maintaining relationships among the data and reducing redundancy. This ability to access the same stored data efficiently from different viewpoints is a major advantage provided by a DBMS as compared to a traditional file system.

2. **Off the Shelf Availability.** A DBMS provides a large amount of complex software already written and debugged. Each DBMS includes one or more high-level access languages, with SQL currently being an informal standard. Because the relational algebra admits a number of potential optimizations, these languages usually support internal efficiencies within the DBMS.[13]

3. **Data Conversion.** Typically, a relational database has some number of supported data types. Typically these include character strings, logical quantities (i.e., true and false), and numbers. Some databases distinguish between unknown and known absent; CAMPS does not yet make any such distinction. Whenever a CAMPS program uses the relational database, data either is returned in some CAMPS-usable form, or data already in CAMPS-usable form is converted to the database internal form. This data conversion is performed differently according to whether the data is numeric or textual. For textual data, there seems little agreement as to whether capitalization should or should not be significant. The CAMPS implementation of the relational model associates case sensitivity with the attribute. That is, when doing a lookup of some string "Foo", whether that matches "FOO" under some attribute depends on whether the attribute considers case important.

   For numeric data, CAMPS uses MKS units. Thus length is in meters, mass is in kilograms, and time is in seconds. The unit for some attribute in the database is associated with the attribute itself. Thus some attribute orbital-altitude might claim to be in "miles." Under most circumstances, the database has functions that convert between MKS units and whatever the database uses. If no units are associated with the attribute, no conversion is performed.

[13] Unfortunately, these access languages are different from each other (even those that are SQL-like). However, all relational DBMS packages provide the same kinds of operations; they differ in syntax and how some constructs are supported. As an example, all RDBMS packages have some way for an application to step through a relation entry by entry, usually according to some specified ordering criteria. The way this stepping is accomplished differs. For example, the Rbase package for PCs uses three pointers. A pointer can be attached to a relation, with ordering and subset selection being accomplished at the time the pointer is initialized. Attribute extraction can be done relative to such a pointer, which is explicitly advanced by a NEXT construct in the application program. Another PC package—dBase—loads relations into areas, each with an associated pointer to a record. The areas can be assigned symbolic names, so that the FOO area's current record's BAR attribute value is referenced by foo->bar. An area's pointer is advanced by making the area "current" and then explicitly moving the pointer. In dbase, ordering and subsetting is associated with the relation not the pointer as in Rbase.

Instantiation from the Database Most objects processed by CAMPS are instances of types of plan-elements. An object stored in the database is instantiated on demand into an internal CAMPS instance by using knowledge in the plan-element hierarchy to retrieve it from the database.

Planning is conducted by manipulating the internal CAMPS plan-element instances. All changes and additions can be stored back (installed) to the database at the end of a session or whenever the user wants to store partial or complete results. Installation is largely the reverse of the instantiation process, again relying on the plan element knowledge hierarchy.

Extensions to the "Traditional" Relational Model

The normal relational database model is not adquate for use in CAMPS. First, knowledge-based systems frequently use a notion of hypothetical world [14] in which alternative assumptions and possibilities can be explored. Second, in having multiple systems sharing a relational database, it is sometimes necessary to have the systems react to changes in the content of the database. Consequently, the CAMPS RDBMS was extended to provide update notification capabilities.

4.8.2 Required RDBMS Features

Each relation in a relational database consists of a set of tuples (where a tuple is a row in a relation). Except for very primitive objects, no single tuple contains all the data for a given object. Because of sharing of data among objects and because one effect of normalization is to multiply relations, data must commonly be drawn (joined) from different relations and assembled into a plan element instance.

All of the functions required for CAMPS would be met by a DBMS that fully supported a relational algebra language, such as SQL from IBM (see [2] for a full description of RDBMS requirements).[15]

For example, relational operators like *Select*, *Project* and *Join* must be provided, perhaps not under those function names. For instance, a *Select* that can specify which attributes to return is a hidden *Project*. Similarly, a *Select* that can specify two relations and an attribute from each relation on which to match is a hidden *Join*.

Comparison operators of both strings and numbers must also be provided with which to form selection criteria, namely: *Greater Than*, *Equal*, *Less Than*. Finally, temporary

[14] CAMPS is also intended to support the usage of the database by the hypothetical worlds mechanism, however, this has not yet been implemented.

[15] Several other dbms support SQL, including ORACLE from Oracle Corporation, Menlo Park, CA. Furthermore, we are fairly confident that INGRES from Relational Technology, Inc. meets the CAMPS requirements, although we are awaiting detailed confirmation. Finally, we have looked into RDBMS for PC sized machines, and find that except for shared access (which is advertised but untested), both Rbase and dbase meet the CAMPS requirements (except in the area of a database dictionary).

relations, or *views*, need to be created as the result of a combination of Select, Project and Join operations. These temporary relations must be available for further Select, Project and Join operations. [16]

4.8.3 Update Notification

Since the relational database has a long persistence, plans stored there may remain of interest for weeks and years. The request to be notified when information is updated similarly has a long persistence. This has two implications: for human users, the agent may not be on-line when the update occurs; for metaplanning agents (agents created to implement some metaplan), these agents do not exist after the planning session terminates. Of course, changes in human staffing of a planning organization also occur.

Our database serves as the entry point for external changes. We are not concerned with how the database gets changed. The actual source, exact destination, and type of input will be domain-dependent, but it merely has to conform to general relational database standards. Our concern is with the need for the system to become aware of relevant changes so that it can react appropriately. CAMPS supports this with database *update notification*. Our assumptions are that the planner (user or metaplanner) knows what types of changes it needs to know about and that this planner is the most qualified to decide how to and whether to react to a change. Consequently, update notification is limited to watching portions of the database at the planner's request, detecting simple changes, placing a notification message on an update-message queue, and making attempts to deliver the message until success is achieved.

Update notification is a three step process:

1. the database is told to watch for certain events;
2. the event occurs, causing the database to compose an update notification message; and
3. the message is delivered to an appropriate recipient.[17]

As planning proceeds, various agents tell the database that if certain portions of the plan change, they should be notified. The database itself maintains relations containing:

1. What events should be watched for.

[16]This requirement is not difficult to meet. CAMPS can easily manufacture dummy permanent relations. However, because a CAMPS session may end without removing the temporary relations, there may need to be some auxiliary procedure for getting rid of relations with a particular name pattern.

[17]The utility of the update notification mechanism can be dramatically increased by observing that the message delivery mechanisms can also act as a notepad for a single user, and as special-purpose memo passing mechanism among several users.

2. Who should be notified should a watched-for event occur.

3. What notification messages are pending delivery to various agents.

The bottleneck of update notification is the process of watching the database. Therefore, our approach tries to minimize the amount of work performed by a database update. Towards this end, we are reluctant to require the database to perform matching on updates to see if events have occurred. Instead, we rely on a relation and/or record marking scheme (see [2] for details).

4.9 Interface

Planning is an inherently distributed application. Monitoring the execution of a plan involves different concerns and is probably physically remote from the planning and replanning function. For "large" plans (i.e, plans whose execution involves a large organization), the planning and replanning functions themselves are likely to be both physically and intellectually distributed. How, then, can partial and complete plans and observations concerning the execution of those plans be communicated among the planners and execution monitors?

Developing plans without communicating them is surely a waste of time. We would prefer that plans be communicated in a form that could be viewed by the recipient according to the recipient's individual concerns. Ideally, the recipient could explore alternative schedules and resource allocations, both to see the consequences of those changes and to understand why the plan is the way that it is.

Unfortunately, the most common means of communicating plans—pieces of paper—can only be viewed as the presenter wished, and cannot be explored in any way other than conversation with the original planner. A conscientious planner in a mature planning organization would probably produce an information package containing the same plan viewed in a number of different ways.

A better way to communicate plans is to store the plan itself in a database (specifically, a relational database). Armed with appropriate graphics programs, the recipient of the plan could produce pictures, graphs, PERT Charts, Gantt charts, or whatever else desired from the plan in the degree of detail the recipient wished, customized to the interest of the recipient.

CAMPS provides a number of tools which enable a user to display data in a number of ways. The user can customize these tools to produce domain specific capabilities. For example, we have introduced a "forms" interface windows in support of the AMPS domain. The forms simulate standard paper forms already used in Air Force mission planning. The forms tend to focus the user's attention on the details of a specific task from a specific point of view. New forms can be added easily and switching between alternative forms is a simple menu operation.

CAMPS provides a flexible and extensible interface. The interface consists of some number of configurations. Each configuration is composed of some number of panes. Each pane is capable of displaying some type of data, e.g., plan element inspector, tables, graphical data, and LISP/text interaction. Plan element inspector panes typically display the slots of a plan element. The user can "inspect" any slot of a displayed plan element via a choice of a menu associated with the plan element inspector pane or with the plan element itself. The table panes display data in some type of tabular form. Typical displays in a table include database contents, a table of constraint instances associated with some plan element, the task-subtask itinerary of a plan, and the metaplan directory. Graphics panes display PERT charts, gantt charts, histograms, and the plan element hierarchy.

With the exception of the interaction window, each pane is scrollable, either through use of the mouse or via a compressed pop-up window (for the graphics panes), and each pane is automatically updated when data it contains changes (i.e., as a result of constraint firing or planning).

Configurations are controlled by a "display manager" which has evolved throughout the course of the project from an overly controlling agent, to a device which will inform the user of the best place to display some type of information, and let the user select the place of his/her choice. Additionally, the display manager controls the display of menus (both pop-up and stationary).

The user is presented with an array of display capabilities for

1. filtering what is being displayed (displaying only a subset of slots of a plan element),
2. determining the format in which to display the data (through selection of a tool such as a pert, gantt, histogram, table), and
3. obtaining synchrony between different panes of a configuration (adjusting the time scale of the histogram or gantt to match each other).

The system also provides "user experience modes" (novice, normal, developer, power), some HELP, and a limited amount of explanation which describes how metaplans are being used and the reason for constraint violations.

5 Using CAMPS in a Domain Application

Thus far, CAMPS has been used to develop two domain application systems: AMPS—an Air Force tactical Mission Planning System, and EMPRESS-II, a NASA space shuttle mission planning system. For each of these domains, there is a high demand for the specification of tasks and their required resources. Additionally, each of these domains exist within a real time environment where information about changes in the environment necessitate changes to partially or competely planned missions (interrelated sets of tasks).

The simplest kind of planning problem for which AI (or knowledge-based) techniques appear necessary has a list of tasks given in the goal state, and furthermore has the property that a complete list of tasks can be effectively enumerated from the planning problem statement. That is, no significant heuristic search is needed to find out what tasks are required in the solution. However, in the planning of the missions described in this section, there is an element of strategy: what goals should be achieved in the near term (a day or two), which targets should be selected, etc. Excluding these strategy decisions, the tasks that constitute a mission plan can be effectively enumerated (a "strike" task against each target, with "support" tasks as appropriate for each "strike"). In this context, the planning problem becomes one of mixed resource allocation and scheduling. Each task requires some set of resources (the selection of which are subject to a number of constraints), and is to be scheduled such that the resources are available.

If the tasks in a "mission" planning problem do not interact, then obviously each can be planned independently. If each mission can be described as a series of slots (or blanks on a standard form) to be filled in, then the form of the planning problem can be simplified to finding a consistent assignment of values to those slots. Simple approaches to solving this kind of planning problem can be made to work, particularly if application-specific problem-solving knowledge is exploited. For example, there is usually some ordering of slots which minimizes the amount of backtracking required in a generate and test algorithm for filling those slots.

For both of the applications described here, typical system usage is envisioned to consist of a user starting up the workstation and logging in to initialize the application system to the particular user type.[18] As part of the application system's construction, the knowledge base, meta-knowledge base, and toolkit are pre-loaded (the formalisms are, of course, integral parts of the CAMPS architecture). As the user begins to use the toolkit, working memory is filled from the external shared relational database. As the user continues to use the tools to manipulate tasks and resources represented in working memory, various meta-knowledge is brought into play to help the user accomplish goals as detected by the toolkit. The correctness and quality of the tasks being manipulated is judged using the knowledge base of plan elements, metaplans, constraints, rules and predicates. At times, the user can request (either implicitly or explicitly) that consistent portions of the working memory be written

[18]There are many categories of user and CAMPS supports four types—novice, developer, normal, and power. For the most part, we describe how a normal user would use the application system, and not how the developer is using CAMPS to develop an application.

back into the relational database so that the results become "visible" to all the users of that database (See [5] and [6] for further details).

CAMPS has been designed to operate in realistic, real-time environments. In the real world, circumstances under which a plan is made can change, either during the planning process or when the plan is being executed. As real-world planning environments contain uncertainty, the success of a plan may be predicated on some condition which cannot be determined at the time the plan is being constructed. However, the real-time requirement precludes the system from reformulating a new plan when new information arrives. CAMPS supports real-time requirements with several mechanisms which allow it to efficiently reformulate existing plans in reaction to changes or new information. These mechanisms have been described in Section 4.8.3 and Section 4.7 and include database update notification, justifications, and hypothetical worlds.

5.1 AMPS—Tactical Mission Planning

AMPS is an Air Force Mission Planning system which has been developed using CAMPS. The purpose of AMPS is to support Air Force tactical mission planners in the developent of tactical fighter mission plans, particularly, offensive counter air (OCA) missions. The domain of AMPS is a decendent of KRS (KNOBS Replanning System) and of KNOBS (KNOwledge Based System)[4, 13] before it.

AMPS' initial domain model was derived almost exclusively from KRS. However, the AMPS domain model has been extended and refined with information derived from a multitude of sources, e.g., monitoring the behavior of planners within the planning environment (at military exercises), and by monitoring the efforts of more applied tactical mission planning systems, such as TEMPLAR[20][19] The OCA is one type of tactical fighter mission. The intention of the OCA is to seek out and neutralize or destroy enemy aerospace forces (e.g., airbase facilities) at some specified time and place. With guidance from the theater commander, information on available assets (aircraft, ordnance), and intelligence information on the state of the battle and the existance of targets, planners propose missions in order to obtain the goals set out in the Commander's guidance. Information about the missions to fly for any particular day are placed on a daily Air Tasking Order (ATO).

The development of the ATO begins approximately 72 hours before the first ATO mission is to fly. During this time interval, plans are devised by the planners to effectively utilize resouces, e.g., aircraft, ordnance, pilots and to support as many missions as possible. Obviously, during this time interval, several environmental conditions are likely to change, e.g.,

[19]TEMPLAR was designed by TRW to serve a specific client, the 9th Air Force. The interface in particular was heavily influenced by the paper forms now used by 9th planners to specify OCA and support missions. TEMPLAR was developed to plan OCA and support missions down to the lowest level of detail that does not require route planning. It was designed to support a medium sized group of planners working in one room together under distributed conditions. That is, more than one planner will be accessing the TEMPLAR databases at the same time, even though these planners will be working on different missions.

weather, asset availability, target locations and validity. Because of this, the planners must continuously monitor these conditions and revise plans as necessary.

Tactical mission planning is a highly dynamic problem solving activity which requires coordination among several individuals responsible for either monitoring or making decisions about different attributes of the overall problem, e.g., intelligence officers monitoring targets, weaponeering officers suggesting ordnance. Communication among the different problem solvers is imperative in order to develop viable plans. In our observations of the planners during military exercises, communication is achieved verbally, through a few automated systems, and by walking paper from organization to organization.

5.1.1 AMPS System Implementation

AMPS is expected to support the tactical mission planning environment, a real-time planning environment. In this section we will described how AMPS has been developed to perform this task.

The purpose of AMPS is to aid the decision making activities of tactical planners in developing tactical missions, particularly Offensive Counter Air missions, in managing resources, in providing information about changes in the environment, and in replanning in response to changes in the environment.

In our view of planning, the planner (human or other) would be interested in monitoring the status of missions, e.g., OCAs. The planner would also be interested in monitoring available assets and the weather at friendly and enemy locations. As AMPS views planning as filling slots subject to constraints, each of these concepts are expressed in AMPS as plan elements with associated slots. Information on the particular aspects of a thing are stored in the database. Thus, for example, information on the weather at particular airbases is stored in a database relation. When a change occurs in the environment, such as to weather, the database will change, and the CAMPS update notification mechanism will detect the change and will inform the user. At a higher level, there are strategies (metaplans) that guide AMPS in selecting which slots to fill first, how to fill them, and what to do if problems are encountered.

The user (a planner) can interact with AMPS through the standard CAMPS interface, or through specialized forms which emulate the paper forms which are used in the planning environment. AMPS contains a set of forms, each tailored to provide pertinant information to a user with a particular need, e.g., asset information, weather, OCA inventory, a target list.

The target planning worksheet is the top level form within AMPS. This form was developed with suggestions from tactical planners at the 9th Air Force. The target planning worksheet is used to plan packages (these contain a related set of missions, such as an OCA with a air escort and/or a SAM suppression mission). Through this form a planner can specify targets (e.g., an airbase), identify target aimpoints (e.g., the runway of an airbase), view and use weaponeering options (ordnance and/or aircraft types as supplied by INTEL), and create new missions.

A planner might begin a planning session by viewing a list of targets for a particular day. For any of these targets, the system can readily inform the user of packages and/or missions which have already been targeted for the specified target. The user can select one of these to work on (perhaps change and replan some aspect of a partially planned mission) or the user can create a new mission for the target. AMPS accesses information from the knowledge base and database in order to support the user.

For any selected mission or for any newly created mission (represented as a plan element), the user can view the slots of the plan element in various levels of detail, filling in the slots which make up the overall plan as he/she chooses. AMPS views a plan as a collection of tasks (a project) arranged into a task/subtask tree. Each task has associated with it temporal and resource requirements. Constraints control the way these slots are filled. The user can fill in slots manually, with some support from the system (e.g., automatically filling slots), or the user can let AMPS complete planning automatically with use of the metaplans. When the user chooses the latter path, AMPS provides a description in the interaction window which describes how metaplans are filling in slots, how resource utilizations for aircraft and ordnance are being established, and how fixit metaplans are being used to resolve any conflicts encountered while planning.

If a constraint is violated, the slots involved in the constraint violation will be backlit and the user will be able to obtain information on why the contraint violation occurred with use of the system menus. Since constraints are declarative, the user can change constraints when the planning environment changes so significantly that current constraints are no longer valid. The user can also invoke the hypothetical planning mechanism in order to explore possible plans, e.g., using a different aircraft.

Task and resource information are stored in the plan element hierarchy and in the relational database. The user can automatically create new tasks when he/she desires to explore a plan in more depth. The user can view task and resource information in a number of ways, e.g., through listings of task/subtask itineraries, through pert charts, gantt charts, and through the histogram. For example, in order to examine how resources are being utilized during some time, by a particular mission and/or other missions, the user can display the resource usage information as a histogram, e.g., the histogram displays F-16 aircraft usage throughout the time when F-16s are being used. The user can adjust the histogram to the time frame of some mission or vise versa in order to see how much of resources of type F-16 are being used by some OCA mission.

The AMPS system is complete with demonstration scenarios and a tutorial which describes how to use the system in either the developer or the user mode.

5.2 EMPRESS

EMPRESS-I and its successor EMPRESS-II are knowledge-based systems which were developed jointly by NASA at the Kennedy Space Center (KSC) and The MITRE Corporation from 1984 to 1988. EMPRESS-I/II support planning and scheduling in the space shuttle payload processing domain by generating preliminary mission schedules, identifying conflicts, monitoring resource utilization, providing graphic support, and enabling users to explore alternative futures.

EMPRESS-I/II were designed to incorporate the tools and problem solving heuristics employed by a NASA expert in the planning and generation of payload processing schedules.[31] EMPRESS-I could generate preliminary schedules for payloads by referring to it's knowledge base of schedules, standard flows (planning strategies), resouces, and requirement dates. Much of the payload processing knowledge of EMPRESS-I was contained in "standard flows". These schedule "templates" were developed by the NASA domain expert (prior to EMPRESS-I development) as a way to quickly generate preliminary schedules. They represent his experience with payload processing to date, and were revised accordingly. With information from the standard flows, rules, a resource tracker and a constraint checker, EMPRESS-I could build schedules, maintain task relations, and manage resources.[25]

EMPRESS-II was developed to extend the functionality of EMPRESS-I and to correct some of the inadequacies identified in the underlying architecture. The domain of EMPRESS-II is essentially the same as that of EMPRESS-I (although the resource information was substantially extended) but the architecture of EMPRESS-II is CAMPS. This architecture enables EMPRESS-II to represent conceptual abstractions (task and resource information) internally in a declarative form. Data (concept instances) are stored in a relational database, thus permitting the exchange of information with other NASA planning tools. The architecture also provides improved constraint checking and resource tracking.

Unlike EMPRESS-I, EMPRESS-II generates tasks for mission payloads as a function of generalized notions of the processing requirements of all payloads (as represented in the plan element hierarchy and in the task template declarations), coupled with specific requirements of a payload, and automatically plans and replans activities with reference to domain specific problem-solving strategies. (The reader will find a more detailed discussion than we present here in [31] and [45].) For example, rather than storing the entire task/subtask structure, EMPRESS-II uses task templates associated with specific types of tasks. A task template says that a task of type "NASA shuttle mission" has as a subtask a task of type "level-I processing." Another task template would say that a shuttle mission has a "fly mission" subtask. Associated with the notion of "fly mission" would be a task template saying that it has a prerequisite of "level-I processing." A task template can have a condition associated with it, so that if the condition is true the template is expanded, while if the condition is false CAMPS knows the template will not be needed. One advantage of the template approach can be seen when the condition is indeterminate at the time the template is processed (that is, we may not know enough about the task to tell whether the template applies). In this case, the template remains attached to the task: the architecture knows that when more information becomes available the template should be reconsidered. As EMPRESS-II missions can have hundreds of subtasks, each with various resouce requirements, the CAMPS task template mechanism was heavily used by EMPRESS-II.

Through the CAMPS architecture EMPRESS-II makes a much richer use of constraints than did EMPRESS-I. For example, the estimated durations for specific tasks are computed as assumptions. [20] The assumption "expect a task's duration to be the historical average duration for that type of task" is represented as a constraint. Additionally, EMPRESS-II has a completely declarative formalism for expressing constraints and has some ability to change constraints "on line." This capability is essential to a dynamically changing environment such as the payload processing domain.

While the EMPRESS-II domain requires a heavy use of update notification (to track space shuttle manifest changes) and hypothetial worlds capabilities (to propose contingency plans and to hypothetically evaluate changes to future manifests or to scarce resouce changes), development of the system concluded before these capabilities were stable enough to be implemented and tested.

EMPRESS-II has demonstration scenarios and a tutorial which describes how to use the system in either the developer or the user mode.

[20] See [6] for a description of how CAMPS utilizes a hierarchical PERT technique as a problem-solving strategy for devising schedules.

6 Lessons Learned

This chapter describes certain capabilities of CAMPS at a more detailed level, emphasizing problems encountered, their solutions, and current limitations.

6.1 The CAMPS Plan-Element hierarchy

In a planning system built on CAMPS, the types of objects that are found in the application domain, and are dealt with by the planner, are represented as *plan-elements*. These include types of tasks, resource pools, resource utilizations, organizations, locations, people, and physical objects. Specific types of plan-elements are usually domain-specific, but these are actually more specialized examples of domain-independent types known to the general CAMPS planner. This characteristic is common in AI reasoning systems and lends itself to the use of a knowledge hierarchy in order to capture the similarities shared by different classes of objects. In CAMPS systems, the plan-element hierarchy fulfills this purpose.

6.1.1 A link that connects several components of camps

Our experience with the plan-element hierarchy has been generally positive. The knowledge hierarchy is the core that connects many components of the CAMPS planning system. It supports the CAMPS slot-filling model of planning, specifying the slots of a plan-element instance, the type of slot (simple or multi-valued; local, class, or remote; optional inverse relationships, etc), and what types of fillers they accept. It also has pointers to the database that guide the instantiation of objects in the database into working memory, or the storing of new or modified objects back to the database. Most constraints are also attached to the hierarchy and automatically posted on new instances. Inheritance supports the easy extension of the hierarchy to include new domain knowledge. The system knows when the hierarchy has been modified and automatically recompiles it before the next instantiation occurs, in order to incorporate the latest changes. All planning objects are represented by plan-element instances. Some, like tasks and resource utilizations, require most of their slots to be filled as part of the planning process. Others, like physical objects, have slots whose values are obtained from the database and usually remain static; when changes do occur, they are more likely to result from external reports rather than internal planning actions.

This representation also supports a dual view of the planning problem that CAMPS is designed for. Planning may be viewed as involving a sequence of tasks that require resources, or as a sequence of resource utilizations that enable certain tasks. Each view has its advantages in different situations. For example, resolving a difficulty in allocating a resource for a particular task may be approached more naturally by considering the planning of the resource utilization rather than the consuming task (even though each points to the other and they share many common slots). CAMPS supports both approaches, mostly by failing to distinguish between them during the uniform processing that it gives to all plan-elements and their slots.

6.1.2 Remote slots

Planning in CAMPS is oriented toward the plan-element instance, its slots, and the constraints that judge the values found in these slots. Naturally, there are also relationships that extend across different plan-element instances. CAMPS handles these situations mostly through the use of remote (indirect) slots. This addresses a problem that is encountered throughout CAMPS. Suppose a **person** is defined with **mother**, **father**, and **maternal-grandfather** slots. The problem is how to handle both the **person**'s **maternal-grandfather** slot and the **person**'s **mother**'s **father** slot. Clearly these both point to the same individual, and it would be wrong to implement this in a manner that would allow two different slots with two different values (even if constraints or demons were (mis)used in an effort to enforce their equality). CAMPS handles this problem by making the **person**'s **maternal-grandfather** slot a remote slot which has **mother** as its "through slot" and **father** as its "target slot." That is, accessing one's maternal-grandfather involves finding one's mother and then finding her father. Remote slots are fairly invisible to the user. Accessing the **person**'s **maternal-grandfather** slot returns a slot variable and its value just as any other access does. It happens, however, that this access will have exactly the same return as accessing the **mother**'s **father** slot. In both cases, the same slot variable is found.

While we are convinced that this is the best approach, there are a number of problems associated with it. One problem is what to do when the through slot is not filled. What if we have information about a grandfather before we learn who the person's parents are? Or, suppose a certain subtask normally has to start at a time that is remotely specified by its superior task, but that it is still reasonable to be able to plan out this subtask as an independent task. To support these cases, we need a way to locally store a remote value when the through slot value that leads to it is unknown. CAMPS supports this by using "postponed remote variables" that are stored locally on a plan-element instance and serve as temporary slots in the absence of the true remote slot. The next problem is that the through slot may be filled in after a postponed remote value has been specified. Later, the through slot value may be deleted. CAMPS has to maintain a watch on through slots and keep the remote slot and its corresponding postponed substitute consistent whenever a value is filled or deleted. Determining what values and constraints belong on which variable can be difficult. Installation into the database also has problems dealing with remote slots. If a through slot is unknown, it is necessary to either skip storing the value or to force the through slot to be filled, perhaps with a newly created instance that is mostly skeletal. Neither solution is entirely acceptable.

Constraint instances involving remote slots pose a particular problem. One difficulty is ensuring that a constraint finds the appropriate argument. Constraint instances in CAMPS are normally assigned their arguments (usually slot variables) at posting time. They are then ready to fire without any further preparation, since the variables do not change even if their values or restrictions do. But, this is not true for remote slots, making it necessary for some arguments to be substituted whenever remote links are made or broken. There can also be a problem in determing which slot a constraint should be written for. When a remote link is made, the postponed and remote variables share constraints. But if it is broken, each sees only the applicable constraints from its local plan-element. There is usually a correct choice of slot for which the constraint should be written. In rare cases, both plan-elements may need the same constraint to support situations where they are planned in isolation.

6.2 Interaction with the relational database

The relational database in CAMPS is used for several different purposes. It is the external repository for information about the domain and the partial or completed results of planning. CAMPS instantiates information from the database into plan-element instances in working memory. The results of its planning can then be installed back into the database. The database is also the link between the planner and the external world. Plans produced by CAMPS are communicated to the intended recipients through the database. External changes to the domain are communicated to the planner by changes that are made to the database by external systems. Finally, an interesting use of the database is its role in the generation and judging of candidates to fill slots in plan-element instances. A particular relation may be the repository for all known objects of a certain type. As such, it can provide an initial listing of candidates for slots that need to be filled with that type of object. A database predicate can be used by a constraint to enforce a relationship between slots that draw their values from the database. In addition to judging the acceptabilty of a candidate, it can also be used to generate all values for one slot that satisfy the specified relationship with any existing fillers of other slots.

6.2.1 Compatibility of CAMPS with a commercial database

The relational database currently used with CAMPS was developed for this project for convenience (and necessity) and was never meant to compete with a commercial database. Its major characteristic is that it runs on the same machine as the planning system and is, in fact, entirely loaded into working memory. We have always envisioned that most real applications of CAMPS would make use of a commercial database and have been careful to adhere to relational database standards and to describe the set of commands used by CAMPS that would have to be supported. It is however conceivable that CAMPS can be used with the present database. The TEMPLAR project, a fielded application in the AMPS and KRS domain, actually took the approach of using a database residing in working memory. Although that system is pushing the limits of RAM and disk memory on a Symbolics 3600 series, it appears to be successful.

The way in which CAMPS uses its database does pose some problems for its use with a separate database. It is unclear how much of the database would in fact be loaded into memory as planning progresses. There are certain actions taken by the planner that involve storing previous database results for future use. It is not certain how this would be done once a clearer separation between the database and planner is made. Perhaps, the planner could store a previous database result and later query the database, passing the old result in as an argument. Alternatively, the planner might need to store a result in a form that it could operate on directly. This would, of course, introduce more of the database into memory (and possibly database operations into the planner). The most obvious use of the database during planning involves the need to draw on the database when instantiating known planning objects into plan-element instances in working memory. This involves essentially one-time actions, since a subsequent search will find the actual instance, but even so there is a need to limit, when possible, the potentially enormous number of objects that might be instantiated during planning; this issue is discussed in the next subsection. Here, we will describe two other activities that also account for intense database activity during planning. These are the use of database predicates and slot candidate generation.

Ordinary constraint evaluation may rely on predicates that are based on the database. These normally check whether certain combinations of values, corresponding to the values found in specified slots, can be found in a particular relationship in the database. Obviously, predicate evaluation is an extremely common action during planning. If the database needs to be queried (across machines) each time such a predicate is checked, then the link to the database must be very fast. Consider what actually happens when a constraint is written to enforce consistency between the selected values for the AIRBASE, UNIT, and AIRCRAFT-TYPE slots of a tactical air mission. This makes sure that the selected unit is assigned to the airbase and possesses the desired type of aircraft. This is the sort of information that is both fairly static and can be found in the database, which makes the use of a database predicate appropriate. (Note that this single constraint and its predicate could easily involve additional slots, making sure that the selected ordnance can be carried by the type of aircraft, that ordnance and aircraft are appropriate for the selected target aim-point, etc.)

The database predicate includes a specification for forming a single relation that contains all known legal combinations of values. Firing this constraint would impose restrictions on each of the three involved slots. Initially, each restriction would bias its slot to select one of the values found by projecting on the relation's attribute that is associated with that slot. If a value is actually selected for one slot, the predicate's relation is changed to one that is formed by selecting on this value. For example, selecting an airbase might restrict the unit to the two units assigned to that airbase and the aircraft type to the three types of aircraft assigned to those units. At all times, each slot is restricted to values found in the current relation associated with this instance of the database predicate. Obviously, we could also use a scheme where restrictions are in the form of various operations on the top-level relations of the database. The CAMPS database is much too slow to compute these operations from scratch during each evaluation of the predicate. A faster database along with delaying evaluation or remembering previous results might allow a cleaner separation between CAMPS's database predicates and the actual database, but it will still require intensive querying of the database.

Another consideration is the way in which the database is used to supply candidates for certain types of slots. Currently, the first time CAMPS needs to look for a candidate for a slot that is to be filled with an instance of a particular type, it creates a generator that points to (the database names of) all of the known instances of that type. This initial generator is stored in memory and is used as a template for all subsequent candidate generators of instances of this type. This avoids the necessity of going to the database every time we consider a new slot that takes the same type of candidates, but it leaves us with two problems. One is the need for the generators to learn about new candidates which are either added to the database or created from scratch in memory; this can be supported without much trouble. The second problem is the need for the generators to frequently give some consideration to each of the possible candidates.

Generating a good candidate starts off with the notion of selecting one possibility, instantiating it into a CAMPS internal value (in the case of relation generators), and then testing this candidate against the known restrictions represented in the generator's filter tree. Ideally, this would look at only one candidate, avoiding the problem under discussion. Unfortunately, it is often the case that the only good candidates are well down the list, or even that no candidate looks particularly promising and we need to find the one that is relatively the best. Therefore, the generator most often tries to prefilter its candidates, ranking them from best to worse by performing low level, relatively inexpensive operations that potentially look at all of the candidates.

Currently, during the creation of the first generator of a given type of instance, camps actually creates a list of all possible values drawn from the relation. During prefiltering, it creates a weighted list, grouping candidates having the same associated belief and disbelief. This is not quite as bad as it might seem. First, restrictions on relation generators are converted to operate on external database values rather than internal instantiated values, so the prohibitive cost of instantiating all candidates is avoided. Second, restrictions often take the form of one belief and disbelief in certain candidates and another belief value that is assigned to "all others." This avoids some of the need to explicitly operate on all candidates. Finally, there is good potential to delay much of the work. The initial candidate list can be replaced by a database operation that provides these candidates. Similarly, the prefiltered weighted list of candidates can also be expressed in terms of operations. Generating the best candidate would involve selecting the operation with the highest belief and executing it againt the initial candidates (expressed as another database operation). This would avoid actually looking at all the candidates. Part of its overhead would result from the possibility that many of the weighted operations would ultimately be discovered to yield the empty set.

6.2.2 The database as a link to the external world

The CAMPS relational database serves as the planning system's link to the external world. The useful results of internal planning end up in the database. Their dissemination relies on standard database operations. Any other system that can use a relational database can access this information for its own purposes. Similarly, external changes are introduced to CAMPS via the database. CAMPS is written with the assumption that the planner will know what sort of changes will need a response. Watches are placed on applicable portions of the database, and changes trigger the formation of messages that are initially stored in the database. Regular attempts are made to deliver these messages to the requested recipients. Watching requested areas and delivering messages are fairly simple procedures and fulfill the responsibility of the update notification process. (It is up to the recipient to update the working memory and to respond to changes if necessary.) This approach relieves the planning system of a lot of burdens. There need be no worries about how information is entered into the database. The developer can also concentrate on the planner inst[illegible] of the external systems that it may be connected to eventually. A simulation of an external system can be as simple as a one line relational database command that modifies a few values in a relation. The events that are triggered by this action are much more interesting than the triggering process.

6.2.3 Working memory and the database

Most of the planning in CAMPS takes place by manipulating objects in the working memory. These objects may be newly created or instantiated from the database. It is not necessarily the case that these objects will be consistent with applicable parts of the database. Changes made by the user/planner will not be seen in the database until they are explicitly installed. This is usually not a problem. The system knows what is in working memory, and others who are accessing the database should not normally see intermediate or tentative results. Changes made by others to the database can have an adverse affect on planning if the working memory does not know about them. But, this is the motivation for update notification. If a change is important, there should be a watch for it which will trigger notification of responsible agents. CAMPS currently supports only one user. If we move to multiple work stations, then these issues will require further attention.

In general, interactions between the working memory and database operate smoothly. Installation into the database has not been tested in an environment where regular daily changes are being made. With our resident database, installation is actually a two step process. Changes are first installed into the relations in working memory. The second step is to save these changes out to appropriate files which store these relations. Developers usually like testing the system against a known state of the world instead of dealing with (and taking the time to manufacture) constant changes. Therefore, most installation that has been performed has avoided the second step of permanently altering the database.

There are also two instantiation issues that deserve attention. CAMPS supports refinement of instantiation type. That is, a request to instantiate requires a unique identifier and a type of object, but it is not necessary for the type to be absolutely specific. A request to instantiate something called `my-car` as a `vehicle` might actually return an instance of type `porsche` (see the section on hypotheticals). If the proper information is available in the plan-element hierarchy and the database, the system will be successful in starting with `vehicle` and refining it to `porsche` (perhaps passing through an intermediate type of `car`). Any refinement should be supportable with a little effort.

CAMPS employs a delayed approach to instantiation. When an object is instantiated, it may have a slot whose value is known to the database. But, if this value is also represented internally as an instance, the instantiation process will simply fill the slot with a "*DB-link*" to the database. It will stay that way until something causes that slot to be accessed, at which time the *DB-link* will be converted into an instantiated value. Additionally, restrictions on a slot that draws candidates from the database are executed in a manner that tries to avoid instantiating the database values. Only the candidate that is actually selected for consideration is instantiated. Despite this delayed scheme, however, the system will sometimes do quite a bit of instantiation in the middle of planning. Some of this is clearly unavoidable, as the system accesses many values during its planning deliberations. Still, at times it may seem to be excessive. Sometimes there is instantiation ripple, where instantiation of an object will trigger instantiation of a parent or subtask, apparently unnecessarily. This typically involves the accessing of remote slots. We may be able to restrict this ripple. But, in any case, no damage is done and these objects usually need to be instantiated eventually.

6.3 Constraint and predicate evaluation

In CAMPS, planning is accomplished by filling slots in plan-element instances. This is normally accompanied by the firing of constraints which evaluate the acceptability of the slot filler. Constraints play the most important role in deciding how a particular slot should be filled. Even when higher level strategies become involved in slot filling, constraints provide important input to them and constraint violations are often responsible for their invocation. The conditions that constraints try to enforce are detected by predicates. Similarly, a constraint will use predicates to test any applicability conditions that it may have. Predicates are evaluated by one or more combinations of three possible means of evaluation. Primitive predicates (which should be domain-independent and found in CAMPS) lead to hand-coded LISP functions. Database predicates check the database for the presence or absence of various values. Finally, rules make use of antecedents which are themselves predicates, eligible to be evaluated by any of the available methods.

6.3.1 Simple constraints lead to large numbers of slots

In KRS, constraints could be checked using arbitrarily complicated LISP code with no limit on possible side effects. That approach can be very tempting to an implementor who is searching for a way of accomplishing a difficult planning action, but it naturally has some serious limitations. The planning system could not be expected to possess much of an understanding of such constraints or to explain their actions to the user who, in turn, would not appreciate having to write such a constraint in order to modify the planning system.

CAMPS is oriented toward constraining slot values using fairly simple constraints. Of the three types of predicates found in CAMPS, those using rules and the database are simple to write and understand. Only hand-coded predicates have any opportunity to be complex. Even here, however, the situation is not comparable to KRS. First, our intention has always been that virtually all hand-coded predicates would be domain-independent and, therefore, located in CAMPS. We want to provide for efficient checking of conditions that are common to many applications and to provide a basic set of primitive predicates for use in domain rules and constraints. But, we want the user of CAMPS to be able to enter simple declarative rules and database predicates rather than writing extensive LISP code, a difficult task which is subject to abuse. Second, the side effects of predicate evaluation should always be limited to the posting of restrictions on a slot's candidate generator. Even when a predicate is fairly complex, the resulting restriction that it posts on a slot is typically much simpler and easier to understand.

In order to support the use of mostly simple constraints, however, most planning systems using CAMPS will probably be found to have more slots in their plan elements and large numbers of simple constraints. These additional slots often will represent what would otherwise be intermediate values in an algorithm employed by a complicated, black box type of constraint. On the positive side, these extra slots tend to represent reasonable planning parameters, and the sophisticated user often appreciates having the option of specifying them. On the other hand, going through the steps of filling these slots is less efficient than passing through an intermediate value in an algorithm. Another advantage to having more slots is that breaking a problem into a greater number of steps makes it easier to solve, understand, explain, and debug. However, this same situation also represents the biggest weakness to this approach. In writing several constraints on several slots in order to enforce a certain relationship among them, it is often easy to lose track of the central, higher level idea that one is trying to enforce.

6.3.2 Effects of slot filling order

The order in which slots are filled can be critical. This is not a new revelation, but the CAMPS approach of having lots of slots (as discussed above) seems to intensify this problem. The problem is simple. Before filling in a slot, the system or user should first try to fill in slots whose values constrain that slot. Without knowledge of these values, constraints on the slot will not have the necessary information to contribute to the selection of a good candidate. Knowing what slot to fill next is, however, a difficult task. As an easy example, consider a constraint that says the duration of a task is the difference between its start and end time. In one situation, it may be possible to calculate the necessary duration based on the type of activity. This would directly determine the finish time once the start is known (or vice versa). However, it is also possible that the start and end will be determined by other means (suppose the task is to wait for something to happen) and will then determine the duration. There is almost never a single correct ordering of slots. The system must be able to determine the correct order in the current situation, taking into consideration, for example, the fact that the user has already set a slot that would normally be filled towards the end when planning from scratch.

6.4 Slot candidate restriction

The most important information associated with a slot is its current fixed value and the constraints that evaluate the acceptability of this value. One prerequisite for automatic slot filling is the ability to find candidates for a slot that agree with the specified content for that slot (ie, they must pass a "type check"). A prerequiste for efficient planning is the ability to select a candidate that will then pass all applicable constraints when tested in the current context. If a serious constraint violation is detected, then it is necessary to change a previously selected value, to decide that the violated constraint does not really apply because of special considerations, or to continue with a flawed plan. Ideally, planning would proceed without any backing up and without serious constraint violations. CAMPS has low level mechanisms that tend to lead to the selection of good values to fill slots. This is accomplished by the use of generators that are associated with each slot. A generator not only produces an initial listing of candidates for the slot, but it is also capable of receiving restrictions that are the side effects of constraint evaluations. By combining these restrictions in a manner reflecting their importance, a generator can produce the candidate that is most likely to satisfy all constraints on its slot.

6.4.1 Look ahead capability

The ability to look ahead, that is, to generate for a slot a candidate that is likely to satisfy constraints, assumed a greater role in CAMPS than originally envisioned. One of the reasons for (results of?) this was our slowness in developing an effective backtracking ability. Dumb (chronological) backtracking in the AMPS domain tends to be very difficult since slots typically have large numbers of candidates. Whenever reasonably practical, it seems easier to put in the extra effort to avoid problems than to be forced to correct them. Even within a generator, filtering individual candidates against restrictions is often a tedious and expensive operation. This is not only because one often has to test a very large number of candidates to find a good one, but also because it is sometimes hard to decide what "good" means. This led to the development of prefiltering, which solves both of these problems by ordering all of the candidates, with their associated beliefs, best first. As it turns out, this can be a comparatively inexpensive operation for candidates derived from number ranges, and even, for some types of restrictions, for candidates supplied by the database.

Our implementation makes use of a "filter tree" which incorporates all of the restrictions on a particular slot derived from the bias-mode evaluation of one or more constraints. This filter tree laboriously combines all of the beliefs, weights, and categories (fixed and calculated) that are associated with constraints and predicates and their rule, database, and hand-coded means of evaluation. We initially discussed many simpler, less brute-force schemes, but none of these appeared to be adequate. The simplicity failed to represent the true relationships between the various restrictions.

6.4.2 Detecting versus avoiding violations

When adding a constraint to a CAMPS system, it is necessary to consider possible differences in detecting a violation and avoiding a violation. Sometimes what at first seems to be the right constraint or predicate for enforcing or checking a relationship will turn out to be wrong for the intended task, as evidenced by the subsequent planning behavior of the system. It is important to understand what a constraint will actually accomplish, as illustrated by the following examples.

Suppose we are planning a `move-box` task and want to ensure that we select a forklift that can safely lift something of the box's weight. A constraint using the predicate `(*greater* ?box-weight ?forklift-rating)` would correctly detect any violation. But this does not directly achieve our stated goal, namely to "select a forklift" that can lift our box. This does not directly influence the selection of the forklift, and altering its capacity once it is selected is probably not a realistic solution. Instead, using this constraint, we would be forced to backtrack and try any other forklifts hoping one will eventually satisfy the constraint. In this case, we really wanted to use a more sophisticated predicate, (*filler-of-filler-greater* ?self :forklift :rating ?box-weight). This says that self (the move-box task) has a :forklift slot whose filler (some forklift) has a :rating slot whose filler should have a value greater than the box-weight. The difference is that this predicate has the capability to post a restriction on the :forklift slot, telling its generator to prefer forklifts with at least a certain weight rating. With this restriction, an acceptable forklift would probably be chosen immediately. Notice that we would need an even more sophisticated predicate, or a second constraint using this one, in order to restrict the selection of the box once the forklift is known. It is not always necessary to look ahead in all situations, but it is a good idea to be aware of what a particular constraint will do.

Suppose we want to write a predicate that checks whether a number is greater than the product of two other numbers. The probable choice would be to write it as a rule-based predicate. The alternative is to write a hand-coded predicate. This would not violate our guidelines, since the predicate is domain-independent. It would also be somewhat more efficient to evaluate. But, it would be much harder to initially write. Writing the rule is quite simple:

```
(Defrule greater-than-product-1
 ((*times* ?product ?mult1 ?mult2)
  (*greater* ?product ?number))
  (greater-than-product ?number ?mult1 ?mult2)
  :belief 1.0
  :cbelief 1.0)
```

There is, however, a difference between this rule and the likely hand-coded alternative. By itself, the rule does not have the capability of restricting all of its arguments. That is, if mult1 and mult2 are known, the predicate can restrict the number slot to be greater than their product; but if number and mult2 are known, the rule has no capability of restricting mult1. Both mult1 and the local rule variable product would be unknown in the first antecedent and the rule evaluation would give up and return no opinion with no side effects. If this restriction capability is desired, then it would be necessary to either write additional versions of this rule or to write a hand-coded predicate that handles all of the cases.

6.4.3 Generation of corrective suggestions

A constraint can be tested in the make-true mode, normally after it has been violated. This mode operates in the same manner as bias mode, except that it attempts to produce suggestions on how to bypass any violations that are found. The advantage of this approach is that it is obtained at almost no additional cost to normal constraint evaluation. Its disadvantage is that the suggestions are generated at a very local level. A violation of (*greater* x y) would produce the suggestions to change the value of slot x to be greater than slot y, or to make slot y be less than x. With the exception of some generated by more complex predicates, most such suggestions are not very helpful unless they are judged by an intelligent planner (human or metaplanning object). For instance, suppose that x is the range of an aircraft and that y is the distance between its points of origin and destination. Obviously, the system must not increase the value in the slot that holds the aircraft's range. Instead, it must, by some means, know that an appropriate action might be to substitute an aircraft having greater range, to lighten the load, add external tanks, or provide in-flight refueling. Similarly, making the distance shorter cannot be accomplished by changing the positions of existing airfields, but it might be feasible to select a different destination or origin, or to insert an extra stop in between. Obtaining this knowledge through reasoning from basic principles can be extremely difficult. CAMPS instead relies on fix-it metaplans that have general or specific knowledge about correcting certain types of problems. A metaplan called to handle a range problem will probably already have enough information available and will not benefit from the additional, low level corrective suggestion. (How does the system know which fix-it metaplan to invoke? We take the easy approach of allowing a fix-it goal to be specified in the declaration of a constraint.) Since it is often the case that an intelligent planner does not need the extra help, the usefulness of these suggestions is restricted to situations where the planner is smart enough to interpret them but dumb enough to need them. CAMPS makes very limited use of this capability, but we consider it an interesting approach with greater potential.

6.5 Problem solving strategies (metaplans and agendas)

CAMPS has a strategic component, represented by metaplan strategies, that provides top-down intelligence to guide planning. Constraints provide sufficient bottom-up knowledge to normally support the automatic filling of a single slot with the best candidate. The metaplanning mechanism supports the automatic planning of the most complex domain tasks. Metaplans translate to an ordered sequence of planning steps. Some metaplans have specific knowledge of how best to plan a particular task, including what slots to fill in what order. Some variations on these metaplans might be particularly applicable for replanning and others might specialize in replanning in time critical situations. Other metaplans are designed to fix problems (e.g., a constraint violation) using knowledge ranging from general planning considerations to very domain specific techniques for fixing specific problems. The strategic component is driven by the posting of goals. Since it is possible that many metaplans might advertise their ability to achieve a particular goal, other metaplans are designed to select the most promising ones to execute first. This decision can be based on which is the most specialized, whether the planning context satisfies applicability filters of some strategies, whether certain global goals have been posted (e.g., conserve certain resources, maximize safety), etc. Metaplans have potential for evaluating the usefulness of a plan and also for explaining what planning decisions were made and why.

6.5.1 Use of metaplans for initial planning

Metaplans ultimately consist of a sequence of steps that are executed in order. Executing a step might do some fairly interesting things, including invoking the execution of other metaplans or replacing itself with a sequence of new steps. One step action is simply to fill a specific slot in a plan-element instance. This first fires constraints in the bias-true evaluation mode, which ensures that all restrictions applicable in the current context are posted on the slot. It then obtains a candidate that best satisfies the combined restrictions of the slot. Finally, after filling the slot with the candidate, it fires constraints again, this time in normal mode (without side effects). The planner makes use of metaplans that advertise their ability to plan a particular task. These metaplans consist of little more than a sequence of slot-filling steps. As long as each step succeeds, the metaplan goes on to execute the next one; upon failure, it posts a fix-it goal and then continues upon success. For example, there is a generic **plan-task** metaplan that advertises its ability to plan any task. It is actually very limited, since it directly only knows about **ESSENTIAL-TASKs** and their slots. Planning of more specific tasks can be accomplished using one of two main alternatives. First, the generic plan-task metaplan has a "get more steps" step. A planner can write a simple metaplan on a more specific task that responds to this request and provides the additional steps needed to plan the specific task. Or, a new plan-task metaplan can be written for the specific task. This metaplan would be selected over the more general one because of its greater specificity. Obviously, this type of metaplan is not particularly intelligent and might not be considered worthy of the "metaplanning" label. In any event, CAMPS has a need for a planning activity that is easily fulfilled using the structure and control mechanisms developed to support metaplanning.

The real problem with using metaplans to specify sequences of slots to fill in order to plan a task is that this approach has significant limitations. In particular, the fixed ordering of slots does not work very well in the general case, as discussed in 6.3.2. The best order depends on the current context. When dealing with time-critical replanning or trying to satisfy certain types of goals, it may be reasonable to provide an alternative metaplan that knows a better way (e.g., sequence of slots) to plan a task. But it does not seem reasonable or very practical to have a different metaplan just because certain slots have already been filled in an order that is somehow considered to be out of sequence. Currently, the planner tends to make do with a single metaplan, sometimes relying on an extra constraint or even backtracking to compensate for this inflexibility. At the same time, we do not envision having the capability of searching the fifty slots of a certain task to find the best one to fill next, filling it, and then repeating the process with the remaining slots. It would, however, be good to use this ability in a more limited manner.

6.5.2 Difficulty of backtracking

General default backtracking in the AMPS domain has not been very successful. Dumb backtracking is easily implemented, but does not produce very good results. One problem is simply that many slots have a very large number of candidates (infinite in some cases) and that the complexity increases very quickly over just a few slots. Remember, that CAMPS's constraints try very hard to look ahead and produce good candidates. Consequently, when a violation does occur, the real cause is typically a value selected several slots previously when the constraints did not have enough information to make a good choice (or were unwilling to do the enormous amount of work needed to check all possible combinations). The desire is to have a smart backtracker that can back up to the point of the error while still retaining relevant information that will guide the selection of a better choice. This is related to the previous paragraph and the problem of determining slot order, which can be based on an understanding of the relationships (e.g., constraints) between slots. For instance, if we leave some of the slots that we back up over filled, this will provide the planner with more context when it selects a new value for the culprit slot. It will, of course, often be totally inappropriate not to unfill intermediate slots. They may, for example, have been directly determined by the slot value that we are now changing. The proper analysis of dependencies could provide us with the needed information.

Most planning failures in CAMPS fall into a relatively small number of categories. Frequently, the constraints that are violated are those that have a particularly large number of slot value arguments. For example, it can be difficult to determine in advance that a particular resource type is not a very good one to select. It may very well be chosen at a time when we have not selected values that will tell us exactly how many we need, how long and at what time we need them, or which suppliers will be in the vicinity of their use. All these values depend on one another and there is usually no ordering that will avoid potential problems. The use of some abstraction may help, but provides no guarantee that movement toward specific values will in fact succeed. The good news, however, is that complicated problems are often accompanied by a good deal of general or specific knowledge about their nature and solution. This knowledge can be incorporated into fix-it metaplans which advertise their ability to find a solution to a particular type of problem. CAMPS relies on these smart fix-it metaplans to control most of the backtracking that is performed by the planner.

6.5.3 Controlling alternative fix-it strategies

When the planner knows about more than one strategy for solving a problem, there is an obvious need to control the process. In general, there should be a way of initially rating the alternatives and then trying the most promising first. As it turns out, this seems to be a situation that can benefit from the hypothesizing capability of the planner. One basic advantage of planning corrective actions hypothetically is that it will no longer be necessary to restore the original state of a violation when a first attempt to solve the problem fails. Instead the planner simply switches to another hypothetical world where the original state of the problem is seen, but where changes dictated by an alternative strategy will affect only the new world. This also gives us much greater flexibility in the problem solving effort. When problems are encountered executing one fix-it strategy, we can simply abandon it for another strategy that now appears to be more promising. If alternatives do not work out, we still have the option of reentering a world and restarting an interrupted fix-it. It may have been suspended before entering a new, more expensive continuation of its basic strategy which now seems to be worth the try. Or, it may have run into constraint violations and the controlling agenda has now decided that debugging this new error is more promising than looking for a more direct solution to the original problem. (Note that all fix-it paths will be controlled by the initial top-level controller. A violation within a fix-it strategy might spawn several alternative solutions for the new conflict, but these are passed to the existing controller so that it can continue to rate all of the potential solutions against one another.) Finally, this approach can be used to show the user the current problem and the results of alternative efforts at a solution. This interaction can either be at the user's request or when the system has failed to find a good solution or is unable to decide between alternatives.

Hypotheticals not only support automatic attempts to fix a conflict, but similarly offer convenient support to the implementor during debugging. Suppose, for example, we are trying to get the planner to resolve a certain type of resource allocation conflict which occurs in a particular scenario after planning ten tasks. Unfortunately, the planner selects what we consider to be an obviously inferior strategy or perhaps it enters a new fixit metaplan strategy and breaks into the debugger fairly disasterously. In either case, it may have made substantial modifications from the original state containing the interesting conflict and it would be nice if we did not have to recreate the original problem from scratch. In this example, however, we are lucky because the problem ocurred while the planner was already in a hypothetical world trying to fix a conflict. All we have to do is abort, reselect the world containing the original problem and reinvoke the planner. The same idea can be used for more general debugging. Before executing something still in need of debugging, simply switch to a hypothetical world before continuing. Now if the planner breaks or doesn't do exactly what we want, we can make a change to the code, return to the world possessing the state of interest, spawn off another world, and then try again, repeating this process as often as necessary.

6.6 Resource tracking

The availability of resources is the single most important limitation on the CAMPS type of planner. If sufficient resources (and time) are available, almost any task can be planned and executed. We have, however, not had much to say about resource tracking simply because CAMPS handles resource utilizations and pools in much the same manner as other plan-element instances. There are, however, a few additional operations (and a lot of code) associated with resource pools as well as some non-slot (ie, not directly controlled by the user) instance variables in utilizations and pools. One problem with resource tracking is the need to support many different domains. We therefore have several different basic types of utilizations and may require more to support the requirements of new domains.

6.6.1 Quantized pools vs. pool sets

CAMPS has two basic types of resource pools. A `quantized-pool` is the actual owner and provider of one type of resource. This might be a pool possessing a single end item (e.g., a particular aircraft which is tracked individually) or a group of like items (e.g., a quantity of fuel, or a number of aircraft that are tracked as a group instead of individually). A quantized-pool knows the type of resource that it possesses, which is normally due to its being defined in the hierarchy as inheriting properties from the pool instance. (In the terminology of Symbolics Lisp Machines, a quantized-pool instance is a "mixin" of both the resource type and the general quantized-pool flavor.) That is, a pool that supplies `F-4C` may actually be an instance of `F-4C` that in turn is built on `AC-POOL`, or it could be an instance of `F-4C-POOL` that is built on both `F-4C` and `AC-POOL`. In either case, it knows that it is an instance of both a type of pool and a type of resource. The normal assigned quantity of the resource is provided by a number found in a slot of the quantized pool. (There is no pre-assigned name for this slot that is applicable for all quantized-pools. Instead, the slot is defined using a `DEFRESOURCE-POOL-QUANTITY` declaration specifying a (usually general) type of plan-element and the name of the slot.) The quantized-pool additionally has a non-slot `reservations` instance variable that holds its confirmed reservations. Finally, it also maintains a time-line on the non-slot `reservability` which shows the quantity of resource available for each time at which a change occurs. This is simply a reflection of its initial availability and the reservations that have been made. When a request is made for a reservation, it checks this time line to verify that it can meet the request during the specified interval. If so, it changes the utilization's status, adds it to the `reservations`, and updates the time-line.

A `pool-set` is another type of pool which in some manner controls quantized-pools for one or more types of resources. That is, a request to support a utilization often does not go directly to a quantized-pool. Instead it is passed to a pool-set which in turn passes the request to a pool that it controls and which possesses the requested type of resource. This allows the consumer to make a more general request (e.g., for two F-4Cs from a certain airbase) and gives the pool-set an opportunity to make an intelligent selection of which pool to task for the request (e.g., a particular squadron at the airbase). The actual quantized-pools are found, directly or indirectly, by looking in the slots of the pool-set instance. The `DEFRESOURCE-POOLS` declaration provides the necessary information which is posted on the applicable plan-element type. For example, `(defresource-pools aircraft air-facility units)` would be used by an airbase to determine that it should look in its `:units` slot if it needs to find suppliers of aircraft. In fact, what it finds are units which are themselves only pool-sets for aircraft. Therefore, `(defresource-pools aircraft unit aircraft-pools)` is then used to discover the actual quantized-pools for aircraft in the unit's `:aircraft-pools` slot.

6.6.2 Types of resource utilizations

A resource utilization is for a quantity of a resource, obtained from a supplier, for use by a consumer, during an interval of time. All utilizations are instances of the plan-element type `resource-utilization` or one of its refinements. Every utilization has `:consumer`, `:resource-type`, `:quantity`, `:supplier`, `:start`, and `:end` slots. The utilization's consumer will normally be a (plan-element instance of a) task which will use the resource. The resource-type will be a plan-element type (a structure that is part of the plan-element hierarchy) describing the type of resource (e.g., `F-16B`, `ELECTRICAL-ENGINEER`, `MOGAS`, etc.). The supplier must be an instance of a `quantized-pool` that is able to supply resources of the specified type. The consuming task and the utilization normally share several slots. If the resource is to be used for the duration of the task, the utilization's start and end time slots will point remotely to the consumer's start and finish slots. For each resource (e.g., aircraft) required to support a task, the task will have a separate slot (e.g., `:ac-reservation`) that holds the utilization. Additionally, it may have remote slots (e.g., `:ac-type`, `:ac-supplier`, `:ac-quantity`) that point remotely to the corresponding slots in the utilization.

There are several different types of utilizations. One basic difference is how the start and end times are determined. The simplest case is when the utilization is for the duration of the consuming task and the start and end times simply point to the start and finish of the consumer. These time slots are, however, not necessarily shared. A reservation for a consumable resource, for example, normally does not expect that resource to be returned. While the start times might coincide, the end time of the usage may be eternity, effectively tying up the resource forever. More realistically, the end time should reflect the expected time of resupply and be capable of later adjustment. In a particular domain, it is not unreasonable to define a different type of utilization for each major resource category. These new utilizations may be built on CAMPS utilization types with little or no change. Frequently, this is done simply to allow the system to reason about a particular type of utilization and to specify different database locations for storing different types of utilizations.

There is a more basic difference between types of resource utilization. The original design for CAMPS envisioned distiguishing between reservations and allocations. A reservation was to represent a lesser degree of commitment between the user and supplier. In effect, it simply provided permission to continue planning a task with the expectation that eventual allocation of required resources could be achieved. The allocation was to represent the firmer commitment that the utilization could definitely be supported barring unexpected events such as maintenance problems. Reservations and allocations are represented by the same utilization; the difference is in the non-slot **status** variable which reflects the status of the utilization (i.e., :incomplete-reservation, :reservation, :allocation). In practice, we have not made use of the distinction between the two types of utilizations. The main difference would be that reservations could be made in an over-booking fashion, where a supplier promises more than it is actually able to deliver during a certain time interval. This could potentially support assessing the actual demand for a particular resource. A pool knows about all of its reservations even if it is over-booked, but it does not have any record of how many requests it rejected. This information is, however, only useful if a request is made to a pool for a good reason. Automatically over-booking may keep the consumer from finding the same resource or an acceptable replacement at an equally convenient alternative pool. Currently, the planner makes use of reservations, does not normally allow over-booking, and consequently does not bother with the final allocation step since there is no real difference. Perhaps a more appropriate difference would be to make reservations that are at a fairly high pool-set level to support more abstract planning. These reservations would not be passed below the pool-set, which would, in effect, act like a quantized-pool that combines all of its subordinate resources of a general type and tracks them as a group. Later, each reservation would be converted to an allocation made with a specific quantized-pool and, perhaps, refined times and quantities. As long as the pool-set has not over-booked, it is likely that most of the reservations could be converted to specific allocations without serious conflict.

6.6.3 Individual vs. group tracking

For some types of resources (e.g., fuel), tracking a large quantity as a group is the natural approach. Even for individual, major end items, group tracking may still be a reasonable alternative to tracking each item in an individual pool. In AMPS, for example, we could track each aircraft individually. This would involve creating an instance of each aircraft which is also its own individual quantized-pool which will track its activity. If we are concerned with tracking the maintenance of each aircraft or knowing which pilots and ground crew are normally assigned to it, then this is really the only approach that works. It is, however, still possible to track a collection of like aircraft (e.g., all those in a particular squadron) as a group. This approach does not support tracking at the same level of detail, but it still is totally consistent. A confirmed reservation with a particular squadron for 4 aircraft does mean that some four aircraft are scheduled to be available at the requested times. This method is more efficient when it can be used. The individual aircraft need not be instantiated or distinguished. The group of aircraft will constitute a quantized-pool with its readily available time-line that immediately reveals whether a request can be supported. When tracking individuals, a squadron would be a pool-set that controls many quantized-pools. A resource request directed to the squadron pool-set would result in a more expensive operation than the comparable request to the squadron quantized-pool. Obviously, the approach used should depend on the needs of the domain.

6.7 Hypothetical planning

CAMPS supports the ability to hypothetically explore different planning alternatives. It uses a coarse, assumption-based approach. At any time, the values seen by the system are consistent with the current world, which is based on a collection of assumptions. These assumptions are obtained by selection from previously defined sets of assumptions. Since the assumptions in a set are mutually exclusive, no more than one assumption from each set is selected to create a given world. Each world has a tag which represents its assumptions. Similarly, a slot variable has a tag, and an operation on it and a world tag determines if it is valid in that world. A variable is actually on a ring of variables representing alternative values valid in different worlds. Finding a valid variable requires rotating the ring to the proper position or creating a new alternative variable if a valid variable is not found. Hypothesizing can be used to explore alternative plans by both the human user and the metaplanning system.

6.7.1 Hypothesizing is explicit and assumptions are atomic

Hypothesizing is an explicit action in CAMPS. Making a change in the current world does not automatically switch to a new world to explore this change. Rather, the default is that the change is made in the current world and any previous state may be forgotten. The user or the planning system must explicitly create assumptions, build a world, and then switch to the world in which the next change should be valid, and out of the world that should retain the previous state. Assumptions are also atomic in that they contain no information about the nature of the assumption except a label. It is possible to switch to a world built on an assumption that "resource X will not be available" and then to promptly select resource X as the one to use. It is assumed that the planner will take steps to make the current world "agree" with its assumptions, but there is currently no way of automatically recognizing or enforcing this. (We could presumably associate with each world a predicate that could check for a simple condition that is intended to characterize that world.) Also, there need not be any inherent mutual exclusiveness between two assumptions in the same set. We are simply stating that anything done by a world making one assumption will not be considered valid in a world making an alternative assumption from that set.

6.7.2 Minimizing the creation of alternative variables

The CAMPS approach is oriented toward limiting the number of variables needed to represent all alternative values in a given slot. Even after extensive hypothesizing has taken place, the typical slot might still have a single value that is valid in all worlds. Naturally, if some world other than the NULL world (which makes no assumptions) wants to modify that variable, then it must be split into a ring with two variables so that only the current world (and those built on a valid superset of its assumptions) will see the modification. As it turns out, the hard thing is not to get the appropriate worlds to believe that the new change is valid, but is rather to ensure that they no longer also see the old value as valid. The initial idea was to add a "splitting assumption" to the tag of the original variable. This would be a special assumption from a set that possesses an assumption made by the current world. Its presence would serve to make it contradict the tag in the current world, but leave it valid in other worlds. Actually, however, it also makes the original variable invalid in any worlds that make an alternative assumption from the same set that has the splitting assumption. This motivated us to make a variable's tag a list, in order to reinstate any of the worlds that made these alternative assumptions. So, a variable is valid if any of its tags is valid in the current world.

6.7.3 Values that can be hypothetical

All CAMPS slots are represented by CAMPS slot variables and all slot variables support hypothesizing by allowing tags and rings of alternative variables. Yet, it appears that some slots should have values that are always valid. For example, the `official-name` of a plan-element instance is a unique identifier and it does not make sense to allow it to vary. Also, consider a task that has a resource utilization that is created to fill a reservation slot. It makes sense that all versions of this task have the same reservation. The actual values (quantity, supplier, etc) may vary by world. But there is no reason that there cannot be a single "reservation for task X" even if it has many different versions. This at least makes for convenient viewing in the interface. After displaying the task and reservation in adjacent inspectors, it is possible to switch among worlds, seeing all valid values for the slots of these two plan-elements for each world.

There are also values that should support hypothetical alternatives but are not CAMPS slots. A resource pool, for instance, maintains a list of all reservations as well as an availability time line representing these reservations. These are not slots since they are not directly modifiable by the planner. Yet, it is clearly necessary for a resource pool to recognize that some of its reservations are hypothetical and not in direct competition with others for the available resources. CAMPS slots are already filled with variables which contain the value of the slot along with about 15 other types of information. Adding a validity tag and also a pointer to a ring of alternative variables for this slot was a simple matter. Non-slot values, however, are represented by flavor instance variables that are set directly to a value. In order to install a hypothetical capability on some of these values, we allow the instance variable to be set to an HVAR (hypothetical variable) that holds the value as well as tag and ring data that support hypothesizing. There are currently no declarations that add a hypothesizing capability to such an instance variable. Rather, all parts of the code that access such a variable are required to use a small set of special macros in dealing with the variable. An HVAR is not even created until some operation requires the splitting of the current value to support hypothetical alternatives.

Since most of a constraint's arguments are slot variables, hypotheticals also affect constraint instances. Copying constraints is one of the more expensive aspects of splitting a variable. Consistency in handling constraints was also the hardest part of the implementation. Lost constraints often were the cause of many planning problems while hypothesizing. One example of the problem occurs when a slot is restricted during constraint evaluation. A restriction constitutes a modification that may be valid only in the current world and therefore may require the variable to split. This means that in the middle of a constraint evaluation, an argument of the constraint becomes invalid which means that the constraint itself is no longer valid. Our technique, when a variable splits, is simply to give it a copy of the list of constraints on the old variable. Then at constraint firing time, a constraint is checked for validity by checking that its posted variables are all valid in the current world. If not, a valid constraint (having valid arguments) is either found on the constraint's ring or a new one is created and added to the ring.

6.7.4 Building worlds on top of other worlds

In a typical application of hypotheticals, the user may be planning in the NULL world and desire to explore two alternative continuations. Suppose we build SOMEC-1 (a "set of mutually exclusive choices") containing assumptions 1A and 1B. Using these we can build world W1A (making only assumption 1A from SOMEC-1) and world W1B. Initially each of these new worlds will see the entire state that is valid in the NULL world (W0). But as new values are set within these new worlds, some variables might split so that they have different values in W0, W1A, and W1B. It makes sense to try out different values in W1A and W1B for some slot that was not set in W0 world. But, what about giving a new value to a slot in W1A which was already set to a different value in W0. This is in fact legal in our system and would result in that slot having two alternative variables. One would have the new value and be valid in W1A while the other would retain the old value and be valid in both W1B and W0.

Now, suppose we wish to explore two additional continuations from the current state in W1A. We could create SOMEC-2 with assumptions 2A and 2B, build worlds W1A2A and W1A2B, and then make various changes in them so that they differ from W1A and one another. We are now faced with the question of what should happen if we switch back to W1A and set a variable that has been set in W1A2A or W1A2B. Or, what effect should making a change back in W0 have on worlds W1A, W1B, W1A2A, and W1A2B? Some users think that these actions should be illegal or that a change made in W1A should not be automatically valid in W1A2A or W1A2B. In any event, our implementation forces a change made in one world to be initially valid in all worlds that are built on a superset of the assumptions made by the world making the modification. For example, changing a variable in W0 will make it valid in all worlds (since all worlds make at least all the assumptions made by W0, which actually makes no assumption). In fact, the ring of variables would be collapsed into a single variable seen by all worlds. It is even possible that this change would supersede an assumption on which a subworld was originally based. Other worlds would, of course, be free to again make changes and split off from the NULL world. We also have the possibility that a particular value valid in world W1A2B could have several potential justifications. It may have been most recently set in W1A2B or W1A or W2B or W0. In any case, it will see the most recent change. Most users who disagree with this implementation would simply prefer that these "conflicts" be illegal. Fortunately, it is a simple matter to avoid them altogether.

6.7.5 Using hypotheticals for contingency plans

Hypothesizing can be used to explore alternative plans by both the human user and the metaplanning system. An additional use might be the creation of contingency plans. It is possible to create various alternatives for each of several plans. Once the planning is done, the system can sit back and monitor external changes. As these changes match some contingencies, it is possible to quickly select a new world consisting of the old assumptions that are still valid and the contingency assumptions that have just become true. This world would, ideally, once again consist solely of valid plans.

Consider the following example using the nomenclature of the previous section. Suppose ten different tasks have been planned for execution on the following day. For each of these tasks, we might explore various things that are likely to go wrong and make hypothetical changes that would respond to these problems. Most of the contingencies would probably address changes of conditions between now and the start of execution. Some might consider changes that occur during the execution. Suppose we create SOMEC-1 with assumptions 1A, 1B, and 1C. Each assumption would assume a particular problem with **task-1**. The assumptions would either be mutually exclusive or we may just be making a practical assumption that at most one of these problems will occur. If necessary, we could use more than one SOMEC for this task. Similary, we might produce a SOMEC for each other task with a few assumptions of various problems that could affect it. For each predicted problem, we could then build a single world, based on a single assumption, and hypothetically plan out a solution to that problem.

At some point, information might be received indicating that the problems corresponding to assumptions 1A, 3C, 7A, and 8D have occured. Even for this relatively simple example, it is obvious that the number of potential combinations of problems is very large. Yet, we need only pop up a menu and build a new world W1A3C7A8D in order to view the current situation. In an ideal case, we will actually see valid changes that address the four reported problems and will need only to disseminate the newest versions of these plans. Naturally, this is probably a very simplistic approach to a difficult problem, but we think that it has potential to at least give us a quick start at replanning in response to the new conditions. There may of course be some conflicts between the various hypothetical solutions. An obvious example would occur if contingencies for several tasks all relied upon a single reserve pool of resources. If too many of these conditions were invoked, the pool could be exhausted. However, this would be represented in the new world by the presence of a resource violation. It would therefore be addressable using normal replanning employing standard resource fix-it metaplanning.

7 The Future of CAMPS as a Tool

The CAMPS architecture is a computer-based AI planning tool that is intended for use in constructing (and, more importantly, modifying) a plan. We intend to develop the CAMPS system with its documentation suite (papers, tutorials, manuals) into a form which developers, other than MITRE will want to use. The resulting tool should have a clean and clearly described architecture, be as close as possible to commercial tools, be readily usable, portable and readily modifiable and extendable in ways in which extenders can readily add their own ideas.

CAMPS is currently equipped with an extensive number of development and debugging tools. Most of these tools enable the user to create, modify, or examine plan elements and their slot contents. Many tools are available throughout the system. While some tools are capability-dependent (database, gantt, plan element), other tools are consistently available, e.g., "inspect-random-thing" which enables the user to view any object selected in the Lisp Machine Inspector Mode. In the following subsections of this section we provide a sampling of how we intend to extend CAMPS' architectural elements to provide a still better development tool.

7.1 Plan Element Hierarchy

The plan element hierarchy is a concept hierarchy consisting of general CAMPS concept capabilities, e.g., essential-task, essential-resource-pool and domain-specific concepts e.g., SAM-site, orbiter-vehicle. A plan element hierarchy tool currently exists which enables the user to graphically view concept elements and AKO relations among these concept elements. As the plan element hierarchy can be very large (larger than a display screen), we have provided a scrolling capability which enables the user to explore the hierarchy by panning over a compressed image of the hierarchy. The user can reduce the complexity of the hierarchy by specifying a node from which a hierarchy can be generated. Obviously, the lower in the plan element hierarchy the element is which was specified, the less information will be displayed. This is very useful, since the hierarchy can be very dense in parts, making it virtually impossible to visually discern relations (represented as lines) between concepts.

For each concept in the hierarchy, the user can get information on what the concept represents. The user can also access the declarative code for any plan element concept through the plan element hierarchy.

Commercial tools (shells) typically provide graphical concept hierarchy tools. We would like to examine the capability provided by these tools in commercial products and modify our existing technique with some of these capabilities. While our use of the plan element hierarchy is typically for validating the declarations of links between plan element concepts, we would like to extend the plan element hierarchy tool to be an instrument for "creating" the plan element hierarchy as well.

7.2 Database Tools

The database is the repository of domain specific information. The database contains information which defines the instances of the concepts defined in the plan element hierarchy. For example, "orbiter-vehicle" is defined in the plan element hierarchy, but "discovery," "atlantis", and other space shuttle orbiter-vehicles are defined in the database. As objects in the environment change (new orbiter vehicles are built), these can be incorporated into the database without impacting the concept plan element hierarchy. However, when new concepts are encountered (heavy launch vehicles for space station) or when old concepts change, then the plan element hierarchy will need to change also.

During the initial development of a application system, developers and knowledge engineers tend to make extensive changes to the database and to the plan element hierarchy, e.g., in the specification of tasks and resources. Existing CAMPS tools enable the developer to examine relations and individual tuples in relations. Relational database information is generally presented to the user in the form of tables. The user is provided with a scrolling capability for examining relations. Since some relations are very large (have many tuples, and/or, have many relation attributes) several filtering tools have been developed which enable the user to: suppress the display of relation attributes, specify search restrictions to reduce the displayable content of the relation table, etc. Tools are provided which also enable the user to add a tuple, modify values, and find hierarchy references.

In our development of AMPS and EMPRESS-II, we tended to develop database relations in the Lisp Machine Editor and to view and test usage of relational database information within the CAMPS environment, particularly in the "database configure" mode. We would like to extend our current database tools to enable the user to build and/or modify database relations within the system environment. While some capabilities are currently available to support this, e.g., modify-value, they have, for the most part, not been used, remaining largly untested.

7.3 Metaplans, Constraints, Rules, and Predicates

CAMPS tools currently exist for getting lists of existing CAMPS and domain-specific metaplans, constraints, rules and predicates; and for modifying these items. For each of these items, the system will provide access to summaries of what the item is supposed to do, as well as an access method for viewing or modifying it in the Lisp Machine Editor.

We would like to provide tools which support the developer, knowledge engineer, and user in the construction of metaplans, constraints, rules and predicates. Currently, much of this is performed in the Lisp Machine Editor support through analogy with existing items. We would like to develop tools which directly support these operations, and we would like to present these tools to the user as part of the CAMPS developer interface.

7.4 CAMPS Interface Tools

When a user instantiates something from the database, or when the user creates a new object, the result can be viewed in the current CAMPS interface through the Plan Element Inspector Windows (which populate several CAMPS configurations). When a plan element has been instantiated, the user can view the current values for each slot; the existing constraints for each slot of the plan element, or for the entire plan element; justification and summaries for each slot which inform the user of who set the slot, and where the slot information is stored (usually a location in the relational database), as well as of other debugging utilities, e.g., examine-slot-restrictions. The plan element inspector windows are also an excellent environment in which to evaluate the relationship of plan element slots with other slots (e.g., indirect slots), and for studying the effects on constraint checking. CAMPS currently backlights slots or slot values which are involved in constraint violations, and provides information about constraint violations through menu items such as slot-constraint-summary.

CAMPS is currently equipped with tools for creating new configurations and for modifying current configurations. Some of these tools are declarative. We would like to support the declarative mode of configuration design as much as possible, building on the CAMPS kernal display mediums which support the generation of: tables, graphics, inspector windows, and the Lisp Machine Listener interaction capabilities.

Extension to some tools is very easy, and some may even be performed within the CAMPS interface. We feel that we have provided the basic tools required for a planning, scheduling and resource allocation environment. These tools include: display tools such as task-subtask displays (both tabular and graphical), pert-charts, histograms, forms, tables, and gantt charts, as well as planning mechanisms such as resource tracking, task template, update notification, and hypothetial worlds.

Finally, since CAMPS is primarily oriented toward the development of planning systems, we feel that it should support, and be judged by its support of the following (see [5] for details):

- *Plan Quality.* If the planning problem can be solved, the tool should be able to find the solution. Furthermore, elements, such as resources should be used "wisely." The promise of AI is that "wisdom" can be encoded and used by a computer program. However, optimality is frequently a chimera; flexibility may be more important than efficiency.

- *Replanning.* Plans change. They are refined, updated, adapted, and modified. Replanning is similar to planning, but different in that in replanning a planner may make choices so as to minimize the resultant perturbations of the original plan. The tool must be able to support all aspects of the planning cycle.

- *Ability to Plan with Incomplete and Inconsistent Information.* We never know enough. Tools must be able to "guess," and distinguish guesses from data. Similarly, the system must be able to distinguish between being ignorant and being confused. Finally, any limitation on the user's ability to produce inconsistent plans should be imposed externally, and should not reflect any inherent limitation of the system to operate in the presence of inconsistency.

- *Explanation of Result.* A "plan" that cannot be explained should not be trusted. A system that cannot explain why decisions were made should also not be trusted. Even less trustworthy is a system that doesn't know what decisions were made. A "plan" is derived and modified by some process. That process itself is subject to question (e.g., "Did you take co-located hazardous operations into account?") and thus must be open to examination and explanation.

- *Dynamic Interoperability.* Planning is never done in isolation, so a tool to support planning must exchange information with other tools supporting other activity. It must both announce its own results (subject to security considerations), and constantly listen to external activity.[5]

References

[1] Carol A. Broverman and W. Bruce Croft. Reasoning about exceptions during plan execution. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, pages 190–195, Seattle, Washington, 1987.

[2] R. Brown and A. Schafer. Relational database utilization by the camps knowledge-based planning architecture: Performance implications. Technical report, The MITRE Corporation, Bedford, Massachusetts, 1987.

[3] R. H. Brown. Agendas: A meta-planning mechanism. Technical Report Corporation M85-26, The MITRE Corporation, Bedford, Massachusetts, 1985.

[4] R. H. Brown, J. K. Millen, and E. A. Scarl. KNOBS: The final report (1982). Technical Report M86-20, The MITRE Corporation, Bedford, Massachusetts, April 1986.

[5] Richard Brown. A solution to the mission planning problem. In *Proceedings of the Second Aerospace Applications of Artificial Intelligence*, Dayton, Ohio, October 1986.

[6] Richard Brown. Knowledge-based scheduling and resource allocation in the camps architecture. In M. Oliff, editor, *International Conference on Expert Systems and the Leading Edge in Production Planning and Control*, Menlo Park, CA, 1987. Benjamin/Cummings.

[7] Richard H. Brown. Allocation of resources in the knowledge-based planning architecture camps. Technical Report M-Series M87-63, The MITRE Corporation, December 1987.

[8] B. Chandrasekaran. Generic tasks in knowledge-based reasoning: High-level building blocks for expert system design. *IEEE Expert*, 1(3):23–30, Fall 1986.

[9] David Chapman. Planning for conjunctive goals. *Artificial Intelligence*, 32:333–377, 1987.

[10] David Chapman and Philip E. Agre. Abstract reasoning as emergent from concrete activity. In M. P. Georgeff and A. L. Lansky, editors, *Reasoning About Actions and Plans, Proceedings of the 1986 Workshop at Timberline, Oregon*, 1987.

[11] Randall Davis. *Applications of meta-level knowledge to the construction, maintenance, and use of large knowledge bases.* PhD thesis, Computer Science Department, Stanford University, 1976. Reprinted in R. Davis and D. B. Lenat (Eds.), *Knowledge-Based Systems in Artificial Intelligence*, New York: McGraw-Hill, 1982.

[12] Randall Davis. Meta-rules: Reasoning about control. *Artificial Intelligence*, 15(3):179–222, 1980.

[13] B. C. Dawson, R. H. Brown, C. E. Kalish, and S. Goldkind. Knowledge based replanning system (krs) final report. Technical Report M87-49, The MITRE Corporation, Bedford, Massachusetts, May 1987.

[14] David S. Day. *Achieving Flexibility for Autonomous Agents in Dynamic Environments.* PhD thesis, Department of Computer and Information Science, University of Massachusetts, Amherst, Mass., 1990. *In preparation.*

[15] Rina Dechter and Judea Pearl. Network-based heuristics for constraint-satisfaction problems. *Artificial Intelligence*, 34:1–38, 1988.

[16] J. DeKleer. An assumption-based tms. *Artificial Intelligence*, pages 127–162, 1986.

[17] R. E. Fikes. REF-ARF: a system for solving problems stated as procedures. *Artificial Intelligence*, 1:127–120, 1970.

[18] Mark S. Fox. *Constraint-Directed Search: A Case Study of Job-Shop Scheduling.* PhD thesis, Carnegie-Mellon University, Computer Science Department, Pittsburgh, PA 15213, 1983.

[19] Mark S. Fox, Brad Allen, and Gary Strohm. Job shop scheduling: An investigation in constraint-directed reasoning. In *Proceedings of the AAAI-82*, 1982.

[20] G. Frekany, M. Imura, P. Lipinski, R. Little, B. Press, L Severin, C Siska, and Z Zayan. Functional description (final) technical expert mission planner (TEMPLAR). Technical Report CDRL A006, TRW Defense Systems Group, Systems Engineering and Development Division, March 1987.

[21] K. Fukumori. Fundamental scheme for train scheduling. AI Memo 596, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1980.

[22] Michael Georgeff and Francois Felix Ingrand. Decision-making in an embedded reasoning system. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 972–978, Detroit, Michigan, 1989.

[23] I. P. Goldstein and R. B. Robert. NUDGE: A knowledge-based scheduling program. AI Memo 405, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1977.

[24] T. R. Gruber and P. R. Cohen. Knowledge engineering tools at the architecture level. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, Milan, Italy, August 1987.

[25] G. B. Hankins, J. W. Jordan, J. L. Katz, A. M. Mulvehill, J. N. Dumoulin, and J. Ragusa. EMPRESS: Expert mission planning and re-planning scheduling system. In *Expert Systems in Government Symposium*, 1985.

[26] D. Krieger and R. Brown. Trimodal interpretation of constraints for planning. In *Proceedings of the SOAR Workshop.* Sponsored by NASA and ASC, August 1987.

[27] T. J. Laffey, P. A. Cox, J. L. Schmidt, S. M. Kao, and J. Y. Read. Real-time knowledge-based systems. *AI Magazine*, 9:27–45, 1988.

[28] Victor R. Lesser, Edmund H. Durfee, and Jasmina Pavlin. Approximate processing in real-time problem solving. *AI Magazine*, pages 49–61, Spring 1988.

[29] Alan K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.

[30] D. McDermott. Planning and acting. *Cognitive Science*, 2:71–100, 1978.

[31] Alice M. Mulvehill. The evolution of a knowledge base. Technical Report M88-45, The MITRE Corporation, October 1988.

[32] James H. Patterson. A comparision of exact approaches for solving the Multiple Constrained Resource, Project Scheduling Problem. *Management Science*, pages 854–867, 1984.

[33] E. D. Sacerdoti. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, 5(2):115–135, September 1974.

[34] E.D. Sacerdoti. *A Structure for Plans and Behavior.* American Elsevier: New York, NY, 1975.

[35] M. J. Stefik. *Planning with Constraints.* PhD thesis, Stanford University, Stanford, California, 1980. Also available as Stanford Computer Science Tech. Report 80-784.

[36] Mark Stefik. Planning with constraints: MOLGEN Part I. *Artificial Intelligence*, 16(2), 1981.

[37] J. P. Stinson, E. W. Davis, and B. M. Basheer. Multiple resource-constrained scheduling using branch and bound. *AIIE Transactions*, pages 252–259, September 1978.

[38] G. J. Sussman. *A Computer Model of Skill Acquisition.* American Elsevier, New York, NY, 1975.

[39] Austin Tate. INTERPLAN: A plan generation system that can deal with interactions between goals. Machine Intelligence Research Unit Memo MIP-1-109, University of Edinburgh, Edinburgh, December 1974.

[40] D. Waltz. Generating semantic descriptions from drawings of scenes with shadows. In Patrick Winston, editor, *The Psychology of Computer Vision*, pages 19–92. McGraw-Hill, New York, 1975.

[41] Robert Wilensky. Meta-planning. In *Proceedings of the First Annual National Conference on Artificial Intelligence*, pages 334–336. Morgan Kaufmann Publishers, Inc., 1980.

[42] Robert Wilensky. *Planning and Understanding: A Computational Approach to Human Reasoning.* Addison-Wesley, Reading, Massachusetts, 1983.

[43] David E. Wilkins. Domain-independent planning: Representation and plan generation. *Artificial Intelligence*, 22:269–301, 1984.

[44] David E. Wilkins. Recovering from execution errors in SIPE. Technical Note 346, Computer Science and Technology Center, Computer Science and Technology Center, SRI International, 1985.

[45] Monte Zweben. CAMPS: A dynamic re-planning system. Technical report, The MITRE Corporation, 1986.

A Filter-trees and Evidence Combination in CAMPS

This appendix discusses in more detail the *filter-tree* data structure in the CAMPS architecture and its use. The discussion is more detailed, especially with respect to the nature of the implementation, than would be appropriate in the body of the text above. This section assumes that the reader has already read Section 4, especially Section 4.4.

A.1 The Generator Filter-Tree

A generator uses a *filter-tree* to represent all of the restrictions imposed on its candidates. A filter-tree potentially has five different types of nodes. Each tree has a single *top* filter-tree node (TOFTN) which serves as the root. A *constraint* filter-tree node (COFTN) always has the TOFTN as its parent. It represents a particular constraint instance which has imposed one or more restrictions on this slot during its evaluation. A COFTN always has a *predicate* filter-tree node (PRFTN) as a child. The PRFTN represents the actual predicate-expression that serves as the constraint's predicate. Its children represent the predicate's means of evaluation. In the simplest case, the predicate might have a hand- coded or database evaluation path. Such a direct evaluation is represented by the *filter* filter-tree node (FIFTN). This contains the actual low-level filter that takes a candidate and returns a belief and disbelief. A PRFTN might also be rule-based, in which case it would have a child that is a *rule* filter-tree node (RUFTN). This RUFTN would itself have one PRFTN child for each rule antecedent that leads to a restriction on this same slot. Its leaf nodes, possibly reached after a series of PRFTN and RUFTN, would be FIFTN that represent direct evaluations.

Constraint evaluation leads to belief values at several different levels. Direct values are returned only by hand-coded and database evaluation paths and are the basis for all other belief values; a rule-based evaluation path returns beliefs partly based on a conjunctive combination of the individual antecedent results; a predicate returns a belief that is a combination of the disjunctive evaluation paths that support it; a constraint returns a result based partly on the beliefs in its predicate and also the combined beliefs in any constraint conditions (also predicates). In addition, both rules and constraints have other fixed weights that influence their results. The overall belief that a constraint violation exists involves the proper combination of all intermediate evaluation results. During this evaluation, many restriction messages may be sent to various slot generators. When a generator selects a candidate it must obtain an overall evaluation using evidence combination in a manner similar to that employed to obtain an overall constraint result.

A.2 Evidence Combination in CAMPS

The FIFTN's filter is the only object in the filter-tree that directly tests the generator's candidates and returns a belief and disbelief. For an exact understanding of what might happen to these values, it is necessary to take a detailed look at evidence combination in CAMPS. While the intent is to understand evidence combination within slot generators, most of the details that follow also apply to evidence combination used to determine constraint violation or satisfaction. There are, however, some important basic differences.

Constraint evaluation:

1. Belief refers to belief that a violation exists.
2. Belief is based on all evaluations triggered by one constraint. Typically, these evaluations involve values found in many different slots.
3. Belief is used to determine whether a particular constraint is violated. The final constraint belief in no way reflects beliefs derived by other constraints. It does not even represent all the fixed weights on the specific constraint.

Slot candidate evaluation:

1. Belief refers to belief that a candidate for one slot is an acceptable value (and will not cause a violation).
2. Belief is based on all restrictions imposed on one slot. Typically, these restrictions may be triggered by the evaluations of many different constraints.
3. Belief is based on an appropriate combination of beliefs contributed by all constraints that restrict the generator.

Evidence combination definitions.

Each belief and disbelief in CAMPS is a value found in the inclusive interval 0 to 1. The sum of belief and disbelief must not exceed 1. Any amount by which the sum is less than 1 is assigned to uncertainty.

DS-COMBINE, taken from Dempster-Shafer, is used to combine evidence believed to be essentially disjunctive. For two belief/disbelief pairs (B1 D1) and (B2 D2), the DS-COMBINE results are:

$$\texttt{Combined belief} = 1 - (1 - B1)(1 - B2)/(1 - (B1D2 + B2D1)),$$

$$\texttt{Combined disbelief} = 1 - (1 - D1)(1 - D2)/(1 - (B1D2 + B2D1)).$$

A.2.1 Evidence combination starts with direct evaluations

In CAMPS, there are three ways in which a predicate might be evaluated. Two of these directly return belief and disbelief values. These are the simplest evaluation beliefs, based solely on direct evaluations. CAMPS supports *hand-coded* evaluations for about 40 different predicates. These are predicates that are domain independent and are evaluated by passing arguments (normally slot variables) to LISP functions. They enforce mostly arithmetical relationships or those specified by basic operations on CAMPS PLAN-ELEMENT structures and instances. The belief and disbelief can be any legal combination returned by the function. For example, the `*GREATER*` predicate returns `(1.0 0.0)` or `(0.0 1.0)` when testing fixed values, since the existence of the desired relationship can be determined with no doubt. The `*FUZZY-EQUALS*` predicate returns varying belief based on how close two values are to each other and the size of the delta argument. If some of the arguments to a hand-coded predicate are from slots that have not yet been fixed, an answer of `(0.0 0.0)` might be appropriate.

Predicates can also be directly evaluated by using a *dbpredicate* that tests the relational database. A constraint writer could ensure that the selected unit is at the selected airbase by declaring an `airbase-unit` database predicate. This would simply point to a certain relation and insist that the (external representation) of the values filling the `airbase` slot and the `unit` slot must both be found in designated attribute positions in at least one common tuple of that relation. A dbpredicate returns beliefs based on the presence or absence of values in a relation, with the exact values depending on the dbpredicate's `belief` and `cbelief` weights. The `belief` represents confidence that the predicate is true given that the proper combination of values has been found in a relation. It is the belief that the database is accurate and also appropriate for testing the predicate. The `cbelief` represents confidence that the predicate is false if the appropriate values are not found. It is the belief that the data is accurate and complete. Normally, the resulting belief from a dbpredicate is `(BELIEF 0.0)` for a match and `(0.0 CBELIEF)` otherwise.

A.2.2 Evidence combination and rule evaluation

The third potential evaluation means for a predicate uses rules. A rule evaluation returns a belief and disbelief based on the evaluation of its antecedents and the beliefs in the rule itself. Each antecedent is itself a predicate, so the beliefs that it contributes to the rule evaluation are ultimately derived from direct hand-coded or dbpredicate results, possibly first passing through other rules. The antecedents are considered to be conjunctive. The evaluation result of each antecedent predicate is combined with the others by minimizing belief and maximizing disbelief that no problem exists (i.e: that a candidate should be selected by a generator or that a constraint is satisfied). This result is the overall belief and disbelief in the rule's antecedent. The final overall rule belief in the predicate (the rule's consequence) is obtained by combining the antecedent result with four fixed rule values.

The `Belief` in a rule $P => Q$ provides the degree to which Q is believed given that P is true. The `Disbelief` value for $P => Q$ provides the degree to which Q is disbelieved given that P is true; disbelief is equivalent to belief in $P => NOTQ$. If these do not sum to unity, then the difference represents uncertainty in $P => Q$, meaning that no conclusion should be made about Q if P is true. It represents a lack of both belief and disbelief in Q. The **CBelief** in $P => Q$ provides belief in $NOTP => NOTQ$. The **CDisbelief** in $P => Q$ provides belief in $NOTP => Q$. Normally, at least one of the *Belief* and *Disbelief* values should be zero, as well as at least one of **CBelief** and **CDisbelief**. (Simultaneous belief and disbelief is supported, but its meaning is somewhat unclear.)

These rule weights provide power and flexibility in writing appropriate rules. They are combined with the antecedent result by using DS-combine for the belief pairs (B1 D1) and (B2 D2) where:

- B1 = antecedent **Belief** * rule **Belief** (i.e., belief is belief in the antecedent times belief that the consequent should be believed given that the antecedent is true),
- D1 = antecedent **Belief** * rule **Disbelief** (i.e., disbelief is belief in the antecedent times belief that the consequent should be disbelieved given that the antecedent is true),
- B2 = antecedent **Disbelief** * rule **CDisbelief** (i.e., belief is disbelief in the antecedent times disbelief that the consequent should be disbelieved given that the antecedent is false),
- D2 = antecedent **Disbelief** * rule **CBelief** (i.e., disbelief is disbelief in the antecedent times belief that the consequent should be disbelieved given that the antecedent is false).

Note that the following rules can potentially confirm Q:

1. A rule with Q as its consequence and **Belief** > **Disbelief**.
2. A rule with $NOTQ$ as its consequence and **CBelief** > **CDisbelief**. **CBelief** in $P =>$ $NOTQ$ means **Belief** in $NOTP => Q$.
3. A rule with $NOTQ$ as its consequence and **Disbelief** > **Belief**. **Disbelief** in $P =>$ $NOTQ$ means **Belief** in $P => Q$.
4. A rule with Q as its consequent and **CDisbelief** > **CBelief**. **CDisbelief** in $P => Q$ means **CBelief** in $P => NOTQ$ means **Belief** in $NOTP => Q$.

The following rules can potentially disconfirm Q:

1. A rule with $NOTQ$ as its consequence and a **Belief** > **Disbelief**.
2. A rule with Q as its consequence and **CBelief** > **CDisbelief**. **CBelief** in $P => Q$ means **Belief** in $NOTP => NOTQ$.

3. A rule with Q as its consequent and `Disbelief` > `Belief`. `Disbelief` in $P => Q$ means a belief in $P => NOTQ$.
4. A rule with $NOTQ$ as its consequence and `CDisbelief` > `CBelief`. `CDisbelief` in $P => NOTQ$ means `CBelief` in $P => Q$ means `Belief` in $NOTP => NOTQ$.

We could do away with negated consequences and just use `Disbelief`. But negated consequences seem the more natural way to think about it. We could also do away with non-zero disbelief. Negated consequences could do the job, except for the possibility of having both belief and disbelief in the same rule, a capability that we are probably better off not having.

A.2.3 Evidence combination and predicate evaluation

A given predicate expression has one or more evaluation means. It may be supported by a hand-coded path, a DBpredicate, and/or one or more rules which contain the predicate in the rule's consequence. These evaluations are discussed above. When a predicate expression has more than one evaluation means available, each is considered to provide beliefs that are disjunctive. The overall belief in the predicate is obtained by combining the result of each evaluated path using `DS- combine`.

A.2.4 Evidence combination and constraint evaluation

A CAMPS constraint is enforced by a predicate expression. It may additionally have applicability conditions (zero or more predicate expressions), and it also has various fixed weights. These all contribute to the final constraint beliefs. If a constraint has no conditions (that is, no predicates are specified in the condition slot of the constraint), the result of testing the condition defaults to absolute belief that the constraint is applicable. When it has one or more conditions, the predicate of each conditions is evaluated in `normal` mode (with no side effects). (It would normally be undesirable to post restrictions in order to make a constraint applicable or not applicable. An exception is in the `make-true` context. One of the ways of correcting a constraint violation might be to post a restriction which forces the violated constraint to become inapplicable.) Each condition is considered to provide conjunctive belief in the applicability of the constraint. Each condition predicate result is combined by minimizing belief and maximizing disbelief in applicability. However, currently the belief in the final condition is then supplemented by transferring to it all of the uncertainty. So the final condition's belief is the result of subtracting the final condition's disbelief from 1.

If the conditions have been satisfied, the constraint predicate is evaluated in `bias-false` (failure motivating) mode, since it tests for the presence of problems which we wish to avoid. This results in overall belief and disbelief values for the predicate. The constraint also has a fixed belief associated with it. The constraint `belief` value represents the belief that there is in fact a violation given that the conditions and predicate are true. The final constraint beliefs are:

```
Constraint Belief    = (* BPredicate BConditions BConstraint),
Constraint Disbelief = (* DPredicate BConditions BConstraint),
```

where BPredicate and DPredicate are the belief and disbelief of the constraint predicate, BConditions is the overall belief in the constraint conditions, and BConstraint is the constraint's fixed belief value. The resulting overall constraint values are the belief and disbelief that a constraint violation exists.

Within a generator, the overall candidate belief, contributed by a constraint, is calculated somewhat differently. As always, the orientation is shifted from belief that a violation exists to belief that a candidate would be acceptable if selected. The calculation of the overall constraint contribution also considers a few additional components. These are the constraint's *endorsement* value and its *consequence-category*. The endorsement is a fixed value that represents the belief that a value is good if it passes the restrictions imposed by a constraint predicate. This complements the constraint's fixed BELIEF value which represents belief that a value is bad if it fails a restriction posted by a constraint. The consequence-category (one of :feasible, :survival, :success, :efficiency, or :assumption) provides weights that reflect the relative importance of the constraint. The final beliefs by the constraint in the acceptability of a candidate are:

```
Constraint Belief    = (* BValue BConditions EConstraint BWeight),
Constraint Disbelief = (* DValue BConditions BConstraint DWeight),
```

where BValue and DValue are the belief and disbelief provided by the constraint predicate, BConditions is the overall belief in the constraint conditions, EConstraint and BConstraint are the constraint's fixed endorsement and belief values, and BWeight and DWeight are the weights associated with the constraint's consequence- category.

A.2.5 Evidence Combination and Overall Candidate Evaluation

When camps fires constraints (e.g., all those on a particular slot), it does not try to calculate an overall combined constraint violation or satisfaction belief. It is interested only in individual constraint violations based on the final constraint beliefs in a violation. (It does additionally consider a violated constraint's efficiency-category to determine how serious the violation is.) Within a slot generator, however, it is necessary to obtain an overall rating for a candidate based on all restrictions imposed on the slot regardless of source. The current method of combining constraint beliefs in a candidate is simply to use DS-COMBINE. This choice may appear flawed on the grounds that constraints are normally considered as conjunctive and not disjunctive evaluations. Many other schemes were considered and eventually rejected. The use of DS-COMBINE appears to work well *if* certain guidelines are followed in the selection of various constraint weighting factors.

The *consequence-category* can be used to ensure that an important failure will not be masked by unimportant successes. With an appropriate consequence-category, high belief in a value, stemming from a constraint with a weak consequence, will be sufficiently weakened so as to have only a small effect on disbelief in a value that is derived from a more important constraint. As a result, two candidates that each fail an important constraint may have slightly different overall values if one scores better with the unimportant constraints, but each result should still be failing.

Assignment of reasonable *endorsement* and *belief* weights will prevent the masking of an important failure by the satisfaction of several other constraints that also have high consequence-categories. A constraint that checks for a serious constraint violation typically might have a high belief value, but it should definitely have a low (probably zero) endorsement value. (There are no brownie points if the left wing is securely attached; just big trouble if it is not.) In this scheme, an important constraint with a high endorsement must be one that would tend to overcome other important (and probably unrelated) constraint violations (sort of an "If God is actively on our side, nothing else matters" constraint). There are relatively few such constraints in most CAMPS domains. Most exceptions fall into the category of endorsing values that must be so by definition. For example, all reservations have a quantity slot. A reservation for an aircraft SCL might absolutely endorse a value of 1, simply because we define the SCL as including the entire configuration for an aircraft. For compatibility reasons, it is convenient to enforce this using a constraint and a standard reservation.

A.5 Filter-Tree Examples

The SHOW-FILTER-TREE function allows the user to examine a description of what restrictions a particular tree is enforcing. It can be used to show the overall and intermediate evidence combination results for different candidates. Each FIFTN node in the filter-tree provides a documentation string describing its basic action, and the function then shows what happens at each node on the way to the tree's root. The following examples have been altered only with line returns and additional comments.

```
#<AC-RESOURCE-UTILIZATION 32069923> :CAPABILITY
#<SIMPLE-VARIABLE 32070011> :CANDIDATES NIL
(0.0 1.0) #<LIST-GENERATOR 32104937>, BELIEF THAT
#<PLAN-ELEMENT F-4C> SHOULD BE SELECTED

;; This is a the AIRCRAFT-CAPABILITY slot of an OCA (actually a remote
;; slot pointing to the CAPABILITY slot in an AIRCRAFT RESERVATION).
;; The result shown is an evaluation of F-4C as a candidate for this
;; slot with the resulting overall beliefs of (0.0 1.0).

(0.0 0.0) #<CONSTRAINT-INST OCA-AIRCRAFT-SUITABILITY OCX6 SATISFIED>
CONSTRAINT BELIEF/ENDORSEMENT: (1.0 0.0),
:SUCCESS WEIGHTS: (0.9 0.9),
CONDITIONS: (1.0 0.0)
(1.0 0.0) #<PREDICATE *CAPABILITY*>
```

```
(1.0 0.0) #<HAND-CODED *CAPABILITY*> 1131
LFILTER-BY-CAPABILITY PROVIDES BELIEF THAT THE CANDIDATE
POSSESSES CAPABILITY AMPS:OCA-AC.
TRUE: (1.0 0.0); FALSE: (0.0 1.0)

;; This constraint wants the candidate to possess an OCA-AC
;; CAPABILITY.  F-4C satisfies this condition.  Notice that the low
;; level absolute belief is changed to (0.0 0.0) since the constraint
;; has an endorsement of 0.0 (it only complains, never compliments).

(0.0 1.0) #<CONSTRAINT-INST AIRCRAFT-CARRIES-SCL OCX6 SATISFIED>
CONSTRAINT BELIEF/ENDORSEMENT: (1.0 0.05),
:FEASIBLE WEIGHTS: (1.0 1.0),
CONDITIONS: (1.0 0.0)
(0.0 1.0) #<PREDICATE *AC-ORDNANCE*>
(0.0 1.0) #<DBPREDICATE 172352061> 1205
LFILTER-BY-RELATION PROVIDES BELIEF IN A CANDIDATE FOUND
(1.0 0.0) OR NOT FOUND (0.0 1.0) IN THE AIRCRAFT-NAME
ATTRIBUTE POSITION OF A RELATION
(CAMPS-RDB:EQUALITY-MEMBER-SELECT (((AMPS:SCL-TYPE AMPS:SCL-E1)))
(CAMPS-RDB:PROJECT ((AMPS:AIRCRAFT-NAME AMPS:SCL-TYPE))
#<RELAT-STRUCT AC-SCL>))
1 VALUE: F-4G

;; This constraint ensures that the aircraft can carry the SCL that
;; has already been selected.  The candidate fails this test which
;; turns out to be a serious violation.  There is absolute belief
;; that it is not feasible for the aircraft to carry the selected
;; SCL.

(0.1 0.0) #<CONSTRAINT-INST AIRBASE-HAS-UNIT-HAS-AIRCRAFT OCX6
SATISFIED>
CONSTRAINT BELIEF/ENDORSEMENT: (1.0 0.1),
:FEASIBLE WEIGHTS: (1.0 1.0),
CONDITIONS: (1.0 0.0)
(1.0 0.0) #<PREDICATE *AIRBASE-AIRCRAFT-UNIT*>
(1.0 0.0) #<DBPREDICATE 172372360> 1202
LFILTER-BY-RELATION PROVIDES BELIEF IN A CANDIDATE FOUND
(1.0 0.0) OR NOT FOUND (0.0 1.0) IN THE AIRCRAFT-NAME
ATTRIBUTE POSITION OF A RELATION
(CAMPS-RDB:EQUALITY-MEMBER-SELECT (((AMPS:UNIT-NAME AMPS:52TFW)))
(CAMPS-RDB:EQUALITY-MEMBER-SELECT
(((AMPS:AFFILIATION-OF-UNIT AMPS:SPANGDAHLEM)))
(CAMPS-RDB:JOIN-ON-ATTRIBUTES
(((AMPS:UNIT-NAME . AMPS:UNIT-NAME)))
#<RELAT-STRUCT UNIT-CHAR>
#<RELAT-STRUCT AC-POOL-CHAR>)))
4 VALUES: F-4E F-4C F-16 F-4G

;; This constraint wants the candidate to be an AIRCRAFT that is
;; found at the selected UNIT and AIRBASE.  It is satisfied by a
;; database check, but receives little credit due to a small
;; endorsement value.

(0.0 0.0) #<CONSTRAINT-INST FOLLOW-INTEL OCX6 NIL>
CONSTRAINT BELIEF/ENDORSEMENT: (0.3 0.8),
:SUCCESS WEIGHTS: (0.9 0.9),
CONDITIONS: (1.0 0.0)
(0.0 0.0) #<PREDICATE INTEL-SUGGESTION>
(0.0 0.0) #<DBPREDICATE 172376202> 1206
LFILTER-BY-RELATION PROVIDES BELIEF IN A CANDIDATE FOUND
(1.0 0.0) OR NOT FOUND (0.0 0.0) IN THE AIRCRAFT-NAME
ATTRIBUTE POSITION OF A RELATION
(CAMPS-RDB:EQUALITY-MEMBER-SELECT (((QUANTITY 4)))
(CAMPS-RDB:EQUALITY-MEMBER-SELECT
(((AMPS:AIM-POINT AMPS:ALLSTEDT-B-CONTROL-RADAR))
(CAMPS-RDB:PROJECT
((AMPS:AIM-POINT AMPS:AIRCRAFT-NAME
QUANTITY AMPS:SCL-TYPE))
#<RELAT-STRUCT INTEL-AIM-POINTS>)))
0 VALUES:

;; This constraint says to select the AIRCRAFT designated for this
;; mission by higher authority.  No such specification was found in
;; the database, so the constraint offers no opinion on the value.

(0.0 0.54) #<CONSTRAINT-INST AIRCRAFT-APPROPRIATE-FOR-TARGET OCX6 NIL>
CONSTRAINT BELIEF/ENDORSEMENT: (1.0 0.3),
:SUCCESS WEIGHTS: (0.9 0.9),
CONDITIONS: (1.0 0.0)
(0.0 0.6) #<PREDICATE AIRCRAFT-APPROPRIATE>
(0.0 0.6) #<RULE AC-FOR-RADIATOR2>
(0.0 1.0) #<PREDICATE WITHOUT-CAPABILITY>
(0.0 1.0) #<RULE WITHOUT-CAPABILITY1>
(0.0 1.0) #<PREDICATE *CAPABILITY*>
(0.0 1.0) #<HAND-CODED *CAPABILITY*> 1132
LFILTER-BY-CAPABILITY PROVIDES BELIEF THAT THE
CANDIDATE POSSESSES CAPABILITY AMPS:F-4G.
TRUE: (1.0 0.0); FALSE: (0.0 1.0)

;; This constraint checks that the AIRCRAFT is appropriate.
;; Apparently the TARGET is a RADIATOR and one of the rules
;; supporting the AIRCRAFT-APPROPRIATE predicate wants such targets
;; to be attacked by F-4GS.  The fact that it is not an F-4G is
;; somewhat mitigated by nonunity RULE BELIEF and CONSTRAINT-CATEGORY
;; WEIGHT.
```

The function SHOW-TERMINAL-WEIGHTED-CANDIDATES displays the overall prefilter beliefs for a specified slot, along with the leaf node contributions to the overall result. The following is a simple example for the FINISH time of an OCA. Apparently, the START time has been fixed and a default EXPECTED-DURATION is also known. There is a strong constraint that insists that the FINISH be later than the START. A weak constraint wants the FINISH to be roughly the sum of the start and expected duration. Note that the second and third ranges in the result start at the preferred time and move away in both directions with linearly decreasing belief, ending with the second belief pair at a value six hours (the delta argument to the *FUZZY-PLUS* predicate) from the preferred value. Values before the START are absolutely opposed. However, CAMPS at this early stage of planning has no objection to putting the end off indefinitely, other than the very weak conflict based on the expected duration. Additional constraints will impose further restrictions as planning continues.

```
#<OCA AMPS:OCX6 32067713> :FINISH
#<CAMPS:SIMPLE-VARIABLE 32079305> :FIXED 2774547552
((0.00999999 0.0)
(([2774547552 12/03/87 14:39:12] [2774547552 12/03/87 14:39:12])))
((0.00999999 0.0)
(([2774547552 12/03/87 14:39:12] [2774569152 12/03/87 20:39:12]))
(0.008612037 0.1387943))
((0.00999999 0.0)
(([2774547552 12/03/87 14:39:12] [2774525952 12/03/87 08:39:12]))
(0.008612037 0.1387943))
((0.008612037 0.1387943)
(([2774569152 12/03/87 20:39:12] +INF)))
((0.0 1.0)
(([2774525952 12/03/87 08:39:12] -INF)))
((0.0 1.0)
(([2774525952 12/03/87 08:39:12] [2774525952 12/03/87 08:39:12])))
:RFILTER RFILTER-BY-> CAMPS:*GREATER* #<FIFTN 172464317>
#<CONSTRAINT-INST ORDERED-START-FINISH OCX6 SATISFIED 32092175>
((1.0 0.0) (([2774525952 12/03/87 08:39:12] +INF)))
((0.0 1.0) (([2774525952 12/03/87 08:39:12] -INF)))
:RFILTER RFILTER-BY-~= CAMPS:*FUZZY-PLUS* #<FIFTN 172464306>
#<CONSTRAINT-INST DURATION-APPROX-EXPECTED OCX6 SATISFIED 32092225>
((1.0 0.0)
(([2774547552 12/03/87 14:39:12] [2774547552 12/03/87 14:39:12])))
((1.0 0.0)
(([2774547552 12/03/87 14:39:12] [2774569152 12/03/87 20:39:12]))
(0.0 1.0))
((1.0 0.0)
(([2774547552 12/03/87 14:39:12] [2774525952 12/03/87 08:39:12]))
(0.0 1.0))
((0.0 1.0)
(([2774569152 12/03/87 20:39:12] +INF)
([2774525952 12/03/87 08:39:12] -INF)))
```